NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PROGRAM FAMILY FOR EXTENDED PRETTY PRINTER

by

Tae Nam Ahn

June 1983

Thesis Advisor: Gordon H. Bradley

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Program Family for Extended Pretty Printer	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June, 1983
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Tae Nam Ahn	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June, 1983 / 13. NUMBER OF PAGES: 74
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program family, pretty printer, standardization, structured form, level structure documentation, object oriented design, reformat

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents a design and partial implementation of a program family of extended pretty printers. Factors that influence the readability (perception) and understandability (cognition) of computer programs are identified. Extensions to the previous pretty printer designs include a capability to selectively display levels of control of a program. In order to accommodate different computer languages (Continued)

Abstract (Continued) Block # 20

and to allow for several secondary functions, a family of pretty printers is designed. This design facilitates easy extension, contraction and modification.

Approved for public release; distribution unlimited.

Program Family for Extended Pretty Printer

by

Tae Nam Ahn
Captain, Republic of Korea Army
B.S., Korea Military Academy, 1975
B.S., Seoul National University, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

ABSTRACT

This thesis presents a design and partial implementation of a program family of extended pretty printers. Factors that influence the readability (perception) and understandability (cognition) of computer programs are indentified, previous work is reviewed, and new solutions are suggested. Extensions to the previous pretty printer designs include a capability to selectively display levels of control of a program. In order to accommodate different computer languages and to allow for several secondary functions, a family of pretty printers is designed. This design facilitates easy extension, contraction and modification.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Programs are written to be read and understood by people. The textual representation should be such that it is easy to read. That is, the representation of the program should be such that it reduces the visual burden on the user and allows him to develop and exploit visual clues to aid in reading. In addition, the text of the program should be designed so that it is easy for the reader to grasp the meaning of the program: that is, the representation of the program should help the reader understand the program.

Fifteen years ago Dijkstra argued that "... our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible." [Ref. 16]. There is an additional conceptual gap between the program spread out in text and how we represent and manipulate the static program and its dynamic process in our minds. Here also we should try to narrow the conceptual gap so that the program is easy to read and to understand.

In the computer science literature, readability and understandability are often used interchangably. Readability is related to physical conditions, for instance, the size, type font, color, and clarity of characters, proper indentations, and the spacing between lines. Understandability is related to psychological conditions, for instance, pattern, memory, logic, and repetition learnings. Precisely speaking,

readability means good perception and understandabilty means good cognition. The system that will be designed in this thesis will seek to improve both readability and understandability by means of reformatting computer programs and presenting the user with alternative representations to aid understanding.

There is evidence to show that readability and understandability of computer programs is an important issue that is directly related to programmer productivity. Although this has been recognized for some time, further improvements in the textual representation of computer programs are possible. This thesis will review the previous work, analyze the remaining problems, and propose new solutions.

II. EXTENDED PRETTY PRINTER

A. BACKGROUND

In a study of commerical programming practices, Elshoff [Ref. 5] found that most programs were poorly written. They were very large, extremely difficult to read and understand, and more complex than necessary. Furthermore, the study determined that programming language usage was poor and inconsistent. The results of the survey by Lientz [Ref. 6] show that the quality of programming is a generally perceived problem. There has been a major effort to improve programming practices. But there still exist many programs that are difficult to read and understand and yet they must regularly be corrected and/or modified.

There are many factors connected with the readability and understandability of a computer program. The reader's familiarity with the program, knowledge of the application area, and own programming style are important factors that are mostly independent of the program [Ref. 4]. This thesis is concentrated on the representation of program text that impacts its readability and understandability. A readable program always seems to exhibit a common set of properties [Ref. 8] [Ref. 9] [Ref. 10]. The program is well commented. The logical structure of the program is constructed from a common set of single-entry single-exit flow of control units. Variable names are mnemonic and references to them localized. The program's physical layout makes the salient features of the algorithm that is implemented stand out [Ref. 14].

Since abstraction is an important mechanism that people use to understand programs, the suppression of details in a program can aid understanding. Modern design methodologies include top down design using stepwise refinement. In this methodology, the programmer designs successive levels of the program. These levels are visible during the design but are often not visible in the final program. The understandability of a program can be improved by making the levels of the program structure visible. It is true that a program may have all these properties and still be unreadable and not understandable; however, the readability and understandability of a program are certain to suffer when it lacks one or more of the the properties [Ref. 14].

B. DEFINITION

Rubin [Ref. 14] defined a pretty printer as follows: "It is a software tool to format programs to make them easier to read and understand." The extended pretty printer can be defined as: a software tool to improve readability and understandability by adding level documentation, commenting and reformatting. These additional extensions to pretty printers will aid people in understanding the program by making more visible the structure of the program and supporting the viewing of the levels of the program. The primary function of an extended pretty printer is to add some level documentation and comments, to insert spaces and linefeeds between tokens - character strings - and to decide where and how to break lines that are too long to fit on the output medium.

C. GOALS

The methods for improving the readability and understandability of a program use a set of specific transformations that can be applied to the program text. The following program tranformations can be done by an extended pretty printer.

1. Reformat

The consistent formating of programs is very important. Elshoff [Ref. 14] said "Just as paragraphing and sectioning help written English, so can indentation, keyword positioning, and logical grouping aid a programming language.". Those jobs can be done automatically by a pretty printer. It will allow the program to be read more easily.

2. Add Level Structure Documentation

In writing about his experments on program comprehension, Shneiderman [Ref. 17] said "Instead of absorbing the program on a character-by-character basis, programmers recognize the function of groups of statements and then piece together these chunks to form ever larger chunks until the entire program is comprehended." This experiment suggests that the level documentation (chunks) of a program will help the understandability of the program.

3. Standardization

Standardization contributes understandability of a program. To understand this, it is helpful to know the source of the expert programming's capacity. The primary piece of direct behavioral evidence for this is Shneiderman's replication [Ref. 26] for programming of Chase and Simon's classic study on memory for chess position

[Ref. 27]. In both these studies, the experts in a particular domain could memorize information from that domain (i.e. a program or a chess position) far better than novices, provided that the information was appropriately structured. If the structure was made random (by shuffling the statements of the program or rearranging the chess pieces), the advantage of the expert would be greatly reduced. That means that the expert has no better memory than the novice, but rather an elaborate knowledge structure in terms of which correspondingly structured items can be very efficiently encoded [Ref. 15].

If this result is extended to programming, it suggests that the expert programmer gets his better knowledge of programs from visible program structure. As noted above, if the textual representation is not structured (e.g. random), the expert programmer will lose part of his capability. People understand something better when they can integrate it with what they already know. From this view, standardization helps people to understand other people's programs more quickly. The visual cues are important in order to unburden the program reader. The final objectives of computer program standards are to ensure consistency, reduce program development and testing time, improve maintainability of programs, and improve changeability of programs [Ref. 12]. Programming standards are not intended to stifle the imagination of programmers. Experiments of Godfrey [Ref. 12] have shown that standards simply remove the drudgery of coding and allow programmers to concentrate more on the problem at hand. It should be noted that the estabishment of standards is a costly process. It should be kept in mind that programming standards are not a panacea for eliminating all poorly written programs. Adherence to these standards will not automatically produce 'good' code [Ref. 12].

There are multiple levels of understanding a program. It is possible to follow each line of code without understanding the overall program function. It may also be possible to understand the program function but not understand each of the steps. There is also a middle level of understanding concerning control structures, module design, and data structures [Ref. 17]. Skimming for a top down view is to suppress detail until the overall program is understood. Then the program is read selectively and understood in more detail.

4. Example

The following example will show how the reformatting, level structured documentation, and the standardization help the readability and understandability of a program.

The bubble-sort algorithm will be introduced for this example [Ref. 18]. The idea of the bubble sort is as follows: "We go through a list comparing adjacent items and exchanging those that are out of order. During such a compare-and-exchange pass, an item moves forward in the list until it 'bumps up against' a larger item." [Ref. 18]. An algorithm language [Ref. 18] and structured FORTRAN will be used for this example.

THE ALGORITHM FOR BUBBLE_SORT :

```
ALGORITHM BUBBLE_SORT
     INPUT N
     INPUT LIST(1:N)
     REPEAT
          NO-EXCHANGES <-- TRUE
          FOR I <-- 1 TO N - 1 DO
               IF LIST(I) > LIST(I+1) THEN
                    TEMP <-- LIST(I)
                    LIST(I) <-- LIST(I+1)
                    LIST(I+1) <-- TEMP
                    NO-EXCHANGES <-- FALSE
               END IF
          END FOR
     UNTIL NO-EXCHANGES
     OUTPUT LIST(1:N)
END BUBBLE-SORT
```

UNFORMATTED FORTRAN PROGRAM FOR BUBBLE_SORT :

```
      INTEGER   LIST(100),I,N,TEMP
      LOGICAL   NOEXG
      READ(5,100) N
  100 FORMAT(I5)
      READ(5,100) LIST
    5 CONTINUE
      NOEXG = .TRUE.
      DO 777 I=1,N-1
      IF(.NOT.(LIST(I).GT.LIST(I+1)) GO TO 10
      TEMP = LIST(I)
      LIST(I) = LIST(I+1)
      LIST(I+1) = TEMP
      NOEXG = .FALSE.
   10 CONTINUE
  777 CONTINUE
      IF(.NOT.NOEXG) GO TO 5
      WRITE(6,200) LIST
  200 FORMAT(1X,I7)
      STOP
      END
```

The following shows some of the possible outputs of an extended pretty printer. Indentation is used to improve readability. Selective display of the levels of the control structure of the program both in FORTRAN and in a generalized programming language is used to support improved understandability. The reader selects the textual representation that best supports his current perceptual and cognitive needs.

LEVEL IA :

```
      INTEGER   LIST(100), I, N, TEMP
      LOGICAL   NOEXG

      READ(5,100) N
      READ(5,100) LIST
      REPEAT
           COMPOUND STATEMENT
      UNTIL NOT NOEXG
      WRITE(6,200) LIST

      STOP
  100 FORMAT(I5)
  200 FORMAT(1X,I7)
      END
```

LEVEL IB :

```
      DECLARATION
      DECLARATION

      SIMPLE STATEMENT
```

```
      SIMPLE STATEMENT
      REPEAT
           COMPOUND STATEMENT
      ENDUNTIL
      SIMPLE STATEMENT

      STOP
      END
```

This shows the first level of bubble sort program. Here only the repeat condition is represented, so the reader of the program can see simply the highest level structure of the program and can understand the overall design of the program more easily. The reader can then select presentations that show additional levels until the completed program is displayed.

LEVEL IIA :

```
      INTEGER   LIST(100), I, N, TEMP
      LOGICAL   NOEXG

      READ(5,100) N
      READ(5,100) LIST
    5 CONTINUE
           NOEXG = .TRUE.
           FOR I = 1 TO N - 1
                COMPOUND STATEMENT
           ENDFOR
      IF(.NOT.NOEXG) GO TO 5
      WRITE(6,200) LIST

      STOP
  100 FORMAT(I5)
  200 FORMAT(1X,I7)
      END
```

LEVEL IIB :

```
      DECLARATION
      DECLARATION

      SIMPLE STATEMENT
      SIMPLE STATEMENT
      REPEAT
           SIMPLE STATEMENT
           DO FOR
                COMPOUND STATEMENT
           ENDFOR
      ENDREPEAT
      SIMPLE STATEMENT

      STOP
      END
```

LEVEL IIIA :

```
          INTEGER   LIST(100), I, N, TEMP
          LOGICAL   NOEXG

          READ(5,100) N
          READ(5,100) LIST
        5 CONTINUE
               NOEXG = .TRUE.
               DO 777 I = 1, N - 1
                    IF LIST(I) > LIST(I+1) THEN
                         COMPOUND STATEMENT
                    ENDIF
      777       CONTINUE
          IF(.NOT.NOEXG) GO TO 5
          WRITE(6,200) LIST

          STOP
      100 FORMAT(I5)
      200 FORMAT(1X,I7)
          END
```

LEVEL IIIB.

```
          DECLARATION
          DECLARATION

          SIMPLE STATEMENT
          SIMPLE STATEMENT
          REPEAT
               SIMPLE STATEMENT
               DO FOR
                   IF CONDITION THEN
                        COMPOUND STATEMENT
                   ENDIF
               ENDFOR
          ENDREPEAT
          SIMPLE STATEMENT

          STOP
          END
```

For most experienced programmers who are familiar with top down design with stepwise refinement, the following representations are easier to read and understand than the intial programs.

FINAL SOURCE PROGRAM :

```
          INTEGER   LIST(100),I,N,TEMP
          LOGICAL   NOEXG
C
          READ(5,100) N
          READ(5,100) LIST
        5 CONTINUE
          NOEXG = .TRUE.
          DO 777 I=1,N-1
               IF(.NOT.(LIST(I).GT.LIST(I+1)) GO TO 10
                    TEMP = LIST(I)
```

```
                    LIST(I)  = LIST(I+1)
                    LIST(I+1) = TEMP
                    NOEXG = .FALSE.
   10         CONTINUE
  777   CONTINUE
        IF(.NOT.NOEXG) GO TO 5
        WRITE(6,200) LIST
C
        STOP
  100   FORMAT(I5)
  200   FORMAT(1X,I7)
        END
```

FINAL STRUCTURE DOCUMENTATION :

```
        DECLARATION
        DECLARATION

        SIMPLE STATEMENT
        SIMPLE STATEMENT
        REPEAT
             SIMPLE STATEMENT
             DO FOR
                  IF CONDITION THEN
                       SIMPLE STATEMENT
                       SIMPLE STATEMENT
                       SIMPLE STATEMENT
                       SIMPLE STATEMENT
                  ENDIF
             ENDFOR
        ENDREPEAT
        SIMPLE STATEMENT

        STOP
        END
```

III. SOME APPROACHES AND VARIOUS OBJECTS

A. SOME APPROACHES

There have been many attempts to improve understandability and readability. The following are typical examples.

1. Neater2

Neater2 accepts a PL/I source program and operates on it to produce a reformatted version. When in the LOGICAL mode, it indicates the logical structure of the source program in the indentation pattern of its output. A number of options are available to give the user full control over the output format and to maximize its utility. [Ref. 19]

2. Prettyprint

It takes as input a Pascal program and reformats the program according to a standard set of pretty printing rules. The pretty printing rules are given i.e., fixed. [Ref. 22]

3. Pascal Program Formatter

Format is a flexible pretty printer for Pascal programs. It takes as input a syntactically-correct Pascal program and produces as output an equivalent but reformatted Pascal program. The resulting program consists of the same sequence of Pascal symbols and comments, but they are rearranged with respect to line boundaries and columns for readability. [Ref. 20]

The flexibility of Format is accomplished by allowing the user to supply various directives(options) which override the default values. Rather than being a rigid pretty printer which decides how a program is to be formatted, the user has the ability to control how formatting is done, not only prior to execution but also during execution through the use of pretty printer directives embedded in the program. [Ref. 20]

4. Contour

It is a program whose purpose is to graphically illustrate a program's structure. It operates by bounding the scope of loops and conditionals by solid (or nearly solid) lines. When compound statements are embedded in other compound statements, one obtains, rather than confusion, a rather pleasant display reminiscent of the contour lines of a topographical map. [Ref. 22]

5. Syntax-Directed Pretty Printer

It is a language independent pretty printer. It is divided into two phases: the grammer processing phase and the program processing phase. A language grammar for the specific language must be provided. It is much easier and quicker to write a grammar for a language than to code a new pretty printer for a specific language. It can work for all structured programming languages, and with minor modifications, can work for other languages. It can handle such problems as comments and error recovery. [Ref. 14]

6. Others

The recent availability of low cost, high quality computer printers allows additional opportunities to improve readability and understandability. Important characters or words can be represented with different fonts: for instance,

the keywords can be represented by bold characters or be underlined to be recognized more easily than other words. This can improve the readability of program.

B. VARIOUS OBJECTIVES

Although the final objective of all approaches is to improve the readability and understandabilty of the program, there are many secondary objectives. The following are typical examples of them:

Teaching structure: An automatic system that checks structure and indentations can help beginning students learn good programming practice. A system that gives clear corrections to mistakes can provide a student with quick feedback. Such a system helps a student to learn structured programming and to learn a set of programming standards.

Standardization in a programming organization: For large software projects with many programmers, program standardization is necessary to help in communication among programmers.

Reformatting for maintenance: There are many programs that are very difficult to read. The maintenance process can be helped if programs can be transformed into a form that is familiar to the maintenance programmers. The scoping capability of an extended pretty printer as described above can also help programmers understand programs they are correcting and modifying.

Automatic corrections: An extended pretty printer can check the indentation of programs, correct indentation errors, and give the user messages explaining the errors.

From the above observations, several common parts of the existing approaches can be found. First, most of the systems are for a specific programming language, for another programming language they would have to be written again. The one exception is the syntex directed pretty printer; for each new language it requires a grammar for each the language. Defining a correct grammar is not an easy task. Second, most of the systems try to make the pretty printer flexible, but the flexibility is limited to a few options and it is not easy to extend the requests. Most constructs of the pretty printers are fixed, but the constructs themselves can be changed e.g. extended or contracted. New structures for indentation can be generated.

IV. PROGRAM FAMILY

A. DEFINITION

Program families are defined by Parnas [Ref. 13] as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. Program families are analogous tc the hardware families promulgated by several manufacturers. Although the various models in a hardware family might not have a single component in common, almost everyone reads the common 'principles of oprations' manual before studying the special characteristics of a specific rodel [Ref. 13].

B. DESIGN METHODOLOGY

Parnas [Ref. 13] shows how module specifications define a family. This is an important guide for selecting the design methcd. Members of a family of programs defined by a set of module specifications can vary in three principal ways.

1. Implementation methods used within the modules.

Any ccmbination cf sets of programs which meet the module specifications is a member of the program family. Subfamilies may be defined either by dividing each of the main modules into submodules in alternative ways, or by using the method of structured programming to describe a family of implementations for the module.

2. Variation in the external parameters.

The module specifications can be written in terms of parameters so that a family of specifications results. Programs may differ in the values of those parameters and still be considered to be members of the program family.

3. Use of subsets.

In many situations one application will require only a subset of the functions provided by a system. We may consider programs which consist of a subset of the programs described by a set of module specifications to be members of a family as well.

As discussed above, there are many primary and secondary objectives for a pretty printer. One approach to these various demands would be to design a large program with many options. Such an approach has several drawbacks: first, the resulting program would be large and necessarily complex, second, for each specific use of the program the unneeded options will most likely impose an unnecessary computational burden. The notion of a program family offers an alternative design. A separate program will be written for different demands, however, all these programs will share a common design and many modules will be common to several family members.

The concept of program families provides one way of considering program structure more objectively. For any precise description of a program family (either an incomplete refinement of a program or a set of specification or a combination of both) one may ask which programs have been excluded and which still remain [Ref. 13]. The criteria of defining modules can be a way to select or distinguish some design methodologies [Ref. 3].

C. PROGRAMMING LANGUAGE FOR OBJECT ORIENTED DESIGN

A design methodology alone is not sufficient to create computer solutions [Ref. 3]. Some features of a programming language can also help in creating good software. In the following table, P. Wegner has categorized some of the most popular languages into generations, along with some of

TABLE I

Programming Language Generation Table

Generation	languages	Period
1ST	FORTRAN I, ALGOL58	1954 - 1958
2ND	FORTRAN II, ALGOL60 COBOL, LISP	1959 - 1961
3RD	PI/I, ALGOL68, PASCAL	1962 - 1970
GAP		1970 - 1980

the language features they introduced:

ADA was developed at the end of the language generation gap, and so has been influenced by contemporary software methodologies. The following figures show the topologies of each generation and ADA. ADA's topology is not flat like those of the previous generations, but rather is multi-dimensional [Ref. 3].

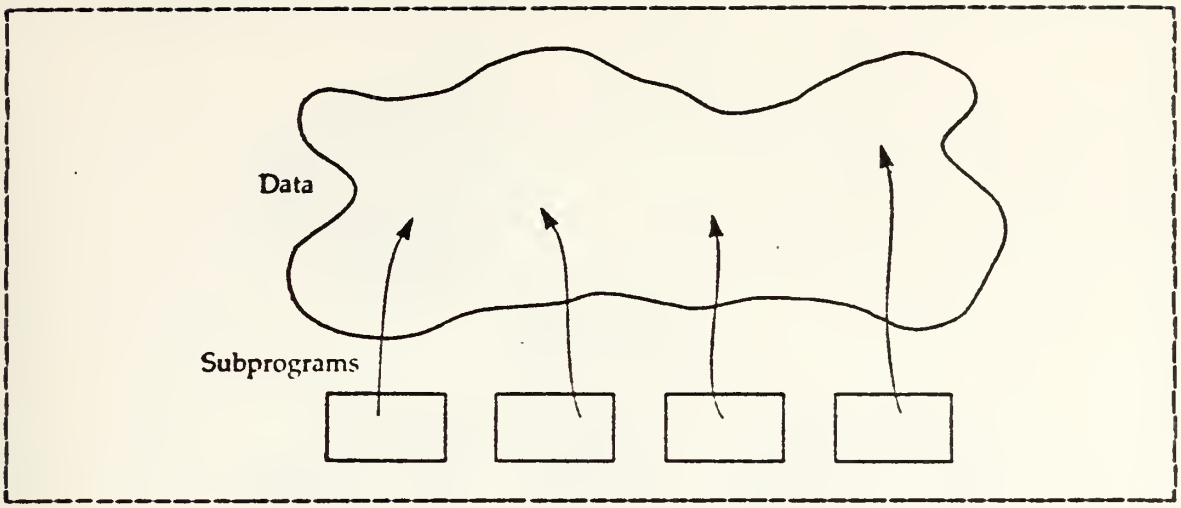

Figure 4.1 Topology for 1st and 2nd Generation.

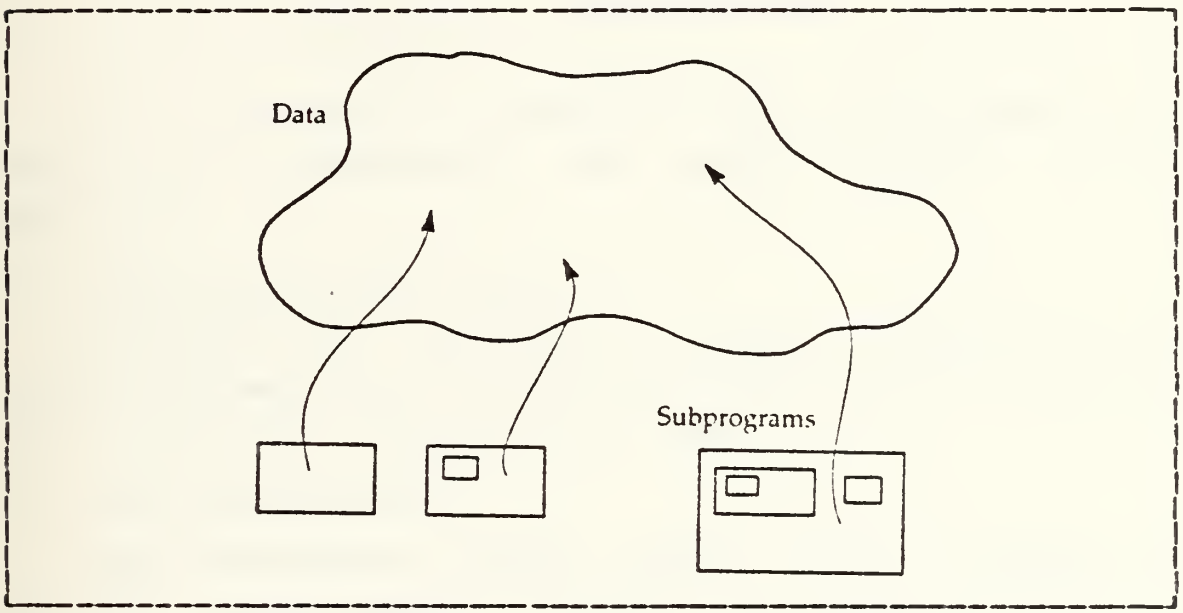

Figure 4.2 Topology for 2st and 3nd Generation.

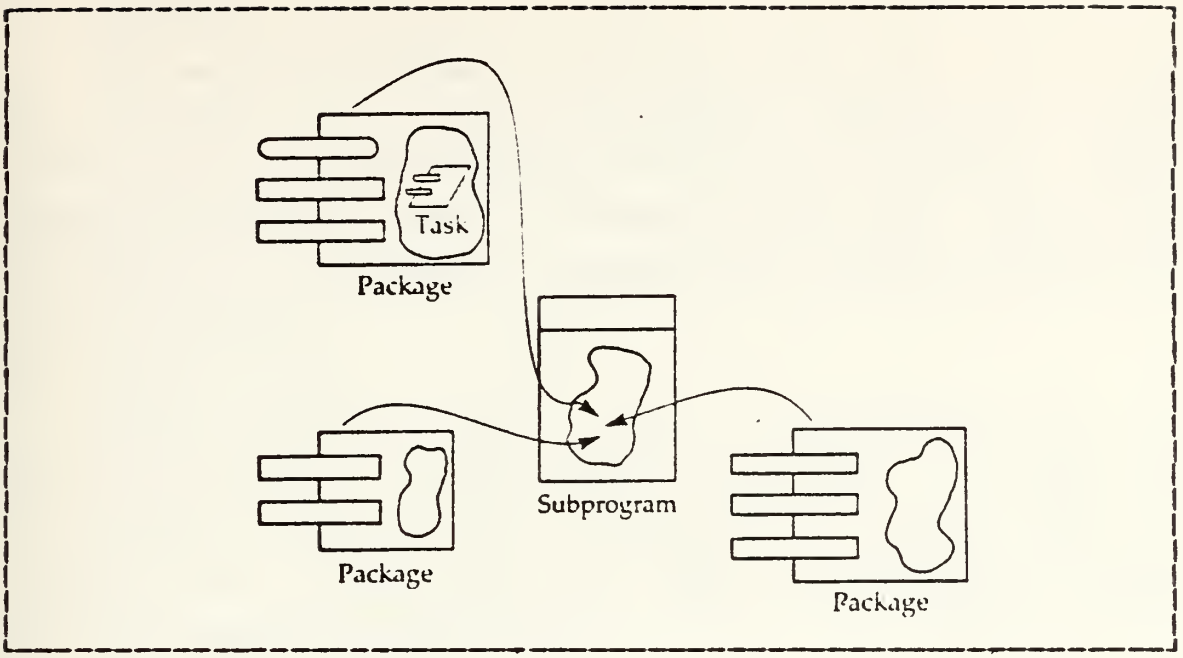

Figure 4.3 Topology of ADA.

The following key features of ADA will support the tools for implementing the object oriented design [Ref. 23].

1. Programming in the large.

Mechanisms for encapsulation, separate compilation, and library management are necessary for the writing of portable and maintainable programs of any size.

2. Exception handling.

Large programs are rarely correct. It is necessary to provide a means whereby a program can be constructed in a layered and partitioned way so that the consequences of errors in one part can be contained.

3. Data abstraction.

Extra portability and maintainability can be obtained if the details of the representation of data can be kept separate from the specifications of the logical oprations on the data.

4. Tasking.

For many application it is important that the program be conceived as a series of parallel activities rather than just as a single sequence of actions. Building appropriate facilities into a language rather than adding them via calls to an operating system gives better portability and reliability.

5. Generic units.

In many cases the logic of part a program is independent of the types of the values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a single template. This is particularly useful for the creation of libraries.

V. MY SOLUTION

A. PROBLEM AND SOLUTION

As shcwn above, most traditional approaches to pretty printers are for a specific programming language. A recent develcpment is the syntax directed pretty printer that can be used for different languages by providing a grammar of the language. The requirement to provide a language grammar represents a non-trivial task. There are many different secondary objectives for a pretty printer for different users. The functions of a traditional pretty printers are not enough to improve both the readability and understandability e.g. the program level construct documentation that traditicnal approaches do not support is needed to help to understand a given program. In short, there are many programming languages and many purposes, but there is not a system that satisfies all those requests and can be modified easily.

In the previous section, the concept of a program family was discussed. The best way to solve the various demands and many programming languages is to construct a program family for the extended pretty printer. The characteristics of program family will permit easy change, easy extensicn, and easy contraction. Each programming language will have a module for itself and data abstraction and procedural abstracticn will be used to hide design decisions that will differ among the members of the program family. Data and procedural abstracticn will also allow some modules to be used by all program family members. For example, the blank operations are a important data abstraction. These oprations can be used for all programming languages and objectives.

B. GENERALIZED PROGRAMMING LANGUAGE CONSTRUCT

For generalized indentation and level documentation, an general internal representation of program structure is required that is independent of any particular programming language. Let us call it a generalized formatter stucture. Since there are many programming language constructs in the many different programming languages, it is too difficult to define a perfect universal programming languge formatter construct. So, we define here a generalized programming language formatter construct that can cover only a limited number of programming languages - structured FORTRAN, PASCAL and some other structured programming languages. For simplicity, the detailed representation of a simple statement will be omitted.

The structure of the program will be shown by indenting the constructs. First, the control structure will be considered. Dijkstra argued that control flow should be limited to three basic structures - linear sequence, structured selection, and structured iteration. But many programmers use the following five structures - if, case, while, until, do for. Also the block can be a element of the structure. Second, most program units are divided into two parts: a declarative part and imperative part. This is also important for the indentation. The Appendix A describes in detail the generalized format structures.

C. ANALYSIS AND DESIGN

1. Analysis

The extended pretty printer has two basic functions. The first is to reformat the source program e.g. indent,insert spaces and linefeeds between tokens and to decide where and how to break lines that are too long to fit on the output medium. The second is to produce level structure documentation of the source program. The basic requirement of the total system is that it has to be easy to change, easy to extend, easy to contract, e.g. it should be independent of the programming language and should be able to fulfill a variety of purposes.

Every structured programming language can be represented as English is - character, word, statement, compound statement(paragraph), unit program(a paper). What is of interest is the way to represent these component as lines. The relationship of these components and lines is very important for the extended pretty printer. The following table represents the relationship of line and statement. The other components have some relation with the statements. So, every component can be represented by lines.

Each level is represented by the source program structures. The structures are represented by statements. So, each statement can have a level degree.

2. Design

As noted in the section on program families, the most important aspect of this system design is to identify the objects. For the indentation, the line and statement are basic elements. Blank is other important object. For the construct representation, level has to be a object.

TABLE II

Relationship Table

LINE	STATEMENT
one	one
one	many
one	part
one	part and one/many

("part" means part of a statement)

The heavily dependent parts should be encapsulated in a module to allow for easy change. The indentation policy can be changed variously, it needs to be manipulated independently. To manipulate the input programming languages independently, the program should be a indepedent module. The program module needs some data structures - STACK, QUEUE -, Keywords table, and some statement oprations. The files - input source file and output file - and their format can be changed easily. So, the input/output files manipulations need be separated from other modules.

For convenience, the module will be divided into two kinds. One is passive modules that are used by other modules but that do not use other modules, for example, blank, level, stack, queue and line. The other kind is active modules that use the other modules, for example, input, output, program and so on. ADA will be used for the detailed design of the system. The following shows the detailed design.

a. Passive modules

(1). Stack Module. This module provides some stack oprations. And it provides the following procedures for other modules that use them [Ref. 24].

```
generic type ITEM is private
package   STACK     is
     type LIST is private;
     procedure CREATE(L: out LIST);
     procedure PUSH(L: in out LIST; I: in ITEM);
     procedure POP(L: in out LIST);
     function  TOP(L: LIST) return ITEM;
     underflow : EXCEPTION;
private   type NODE;
     type LIST is access NODE;
     type NODE is record
          head : ITEM;
          tail : LIST;
     end record;
end STACK;
```

(2). Queue Module. This module provides some QUEUE oprations. And it provides the following procedures for other modules that use them [Ref. 24].

```
generic type ITEM is private;
package   QUEUE     is
     type LIST is private;
     procedure CREATE(L: out LIST);
     procedure ENQUEUE(L: in out LIST; I: in ITEM);
     -- Insert the item into the rear of QUEUE
     procedure DEQUEUE(L: in out LIST; I: out ITEM);
     -- Delete the item from the front of QUEUE
     underflow : EXCEPTION;
```

```
private   type NODE;
     type LIST is access NODE;
     type NODE is record
          head : ITEM;
          tail : LIST;
     end record;
end QUEUE;
```

(3). Blank Module. This module provides all blank operations that insert, remove, count and so on for other modules that need the blank operations.

```
generic type INPUT is private;
package   BLANK     is
     BLK  : constant CHARACTER := ' ';
     type NUM is NATURAL;
     procedure INSERT(N,M: in NUM; P: out INPUT);
     --    N : The start column of a line
     --    M : The number of blanks to be inserted
     procedure DELETE(N,M: in NUM; P: out INPUT);
     --    N : The start column of a line
     --    M : The number of blanks to be deleted
     procedure START(L: in INPUT; N: out NUM);
     --    N : The number of blanks in a line
     --        from the start column
     function IS_BLANK(C: in CHAR); return BOOLEAN;
     --    Check the input character is blank
     --    If blank, return TRUE
     --    Else, return FALSE
     overflow : EXCEPTION;
end BIANK;
```

(4). Level Module. This module will provides the level operations for other modules that need them. The operations are:

```
package   LEVEL     is
```

```
     type NUM is NATURAL;

     procedure INCREASE(L:in out NUM);
     --    Increase the level
     --    L : input/output level number

     procedure DECREASE(L:in out NUM);
     --    Decrease the level

     procedure ZERO(L:in out NUM);
     --    Make the level zero or starting point.

     overflow : EXCEPTION;

     underflow : EXCEPTION;

end LEVEL;
```

(5). Line Module. This module manages the line object. It provides a set of procedures available to other modules that use the line.

```
Generic type LINETYPE is private;

package   LINE      is

     type LINEPOINT is private;

     type NUM is NATURAL;

     type CHAR is CHARACTER;

     procedure GET_LINE
     (P: in out LINEPOINT; L: out LINETYPE);
     --    Get a whole line into internal structure
     --    P : ID for a line
     --    L : Content of a line

     procedure PUT_LINE
     (P: in out LINEPOINT; L: in LINETYPE);
     --    Put the a internal line into the linetype
     --    P : ID for a line
     --    L : Content of a line

     procedure LINE_LENGTH
     (P: in LINEPOINT; N: out NUM);
     --    Compute the line length
     --    P : ID for a line

     procedure GET_CHAR
     (P: in LINEPOINT; N: in NUM: out CHAR);
     --    Get a character that is in given line and
     --    position
     --    P : ID for a line

     procedure PUT_CHAR
     (P: in LINEPOINT; N: in ITEM: in CHAR);

     --    Put the given character into the position
     --    and the line given
```

```
     --   P : ID for a line
     --   N : Column of the line

     procedure FRONT_INSERT
     (P: in out LINEPCINT; L: in LINETYPE);
     --   Insert line in front of the given
     --                  line position
     --   P : ID for a line
     --   L : Content of a line

     procedure REAR_INSERT
     (P: in out LINEPOINT; L: in LINETYPE);
     --   Insert the line at rear of
     --   the given line position
     --   P : ID for a line
     --   L : Content of a line

     underflow : EXCEPTION;

     overflow : EXCEPTION;

private   type NODE;

     type LINEPOINT is access NODE;

     type NODE is reccrd

          content   : LINETYPE;

          front     : LINEPOINT;

          rear      : LINEPCINT;

     end record;

end LINE;
```

(6). Symbol Table Module. This module will manage a symbol table. It is designed for general symbol manipulation.

```
Generic type ITEMTYPE is private;

package   SYMBOLTABLE    is

     N : constant =: 200;      -- size of symbol table

     ITEMSIZE: constant =: 20;

     type ITEM is new STRING(1..ITEMSIZE);

     procedure ADD(X:in ITEM; I: in ITEMTYPE);
     --   Insert an item and the information
     --   associated with it into SYMBOLTABLE

     function IN_TABLE(X:in ITEM) return BOOLEAN;
     --   Check to see if an item is in
     --   the SYMBOLTABLE

     function GET(X:in ITEM) return ITEMTYPE;
```

```
     --   Retrieve the information associated
     --   with an item in the SYMBOLTABLE

     function FULL return BOOLEAN;
     --   Determine whether or not the SYMBOLTABLE
     --   is full

     procedure CLEAR;    -- empty table
     --   Reinitialize (reset) the SYMBOLTABLE

end SYMBOLTABLE;
```

b. Active modules

(1). Input Module. This module hides the input format. It reads the original lines from the input media and calls procedures provided by the line module to store the lines inside of the line object.

```
with TEXT_IO;
with LINE;
generic type LINEPOINT    is   private;
package INPUT is
     type INFILETYPE : TEXT_IO.FILE_TYPE;

     procedure READFILE
     (INFILE: in INFILETYPE; START : out LINEPOINT);
     --   Read the input file and store each line into
     --   internal line structure using LINE module
     --   INFILE : The input file that have source program
     --   START  : The starting line ID of internal
     --            structure

end INPUT
```

(2). Output Module. This module will hide the outfile media. And it will output the indented results, the construct form of the input program and the input using other modules - indent, line and so on.

```
with TEXT_IO;
with LINE;
```

```
with INDENT;
generic type LINETYPE    is   private;
package OUTPUT is
     type OUTFILETYPE  : TEXT_IO.FILE_TYPE;
     type CODEFILETYPE : TEXT_IO.FILE_TYPE;
     procedure PRINT_OUTFILE
     (OUTFILE: out INFILETYPE; START : in LINEPOINT);
     --   Print the indented output into OUTFILE using
     --   indent and line modules
     --   OUTFILE : The output file that has
     --             the indented source program
     --   START   : Line start ID of internal structure
     procedure PRINT_CODEFILE
     (CODEFILE: out INFILETYPE; START : in LINEPOINT);
     --   Print the code documentation using line and
     --   indent module
     --   CODEFILE : The output file that has
     --              the code documentation
     --   START    : Line start ID of internal structure
end OUTPUT
```

(3). Statement Module. This module manages the statement object and also provide a set of procedures available to other modules that use the statement object by using line module procedures.

```
With LINE;
generic
      type INDENTPOINT is private;
package   STATEMENT is
     type NUM is NATURAL;
     type CHAR is CHARACTER;
     type INDENTPOINT is access INDENTNODE;
     type STATEPOINT is access NODE;
     type NODE is record
          content   : STATETYPE;
          front     : STATEPOINT;
          rear      : STATEPOINT;
     end record;
```

```
    type STATETYPE is record
         from            : POSITION;
         to              : POSITION;
         information     : INDENTPOINT;
    end record;
    type POSITION  is record
         line       : LINEPOINT;
         column     : NUM;
    end record;
    procedure GET_STATE_DELIMITOR(D: in CHAR);
    --   Get statement delimitor

    function END_OF_STATE(D: CHAR) return BOOLEAN;
    --   Check the end of a statement

    procedure GET_STATE
    (P: in out STATEPOINT; L: out STATETYPE);
    --   Get a statement using LINE module

    procedure PUT_STATE
    (P: in out STATEPOINT; L: in STATETYPE);
    --   Put a statement using LINE module

    procedure STATE_LENGTH
    (P: in STATEPOINT; N: out NUM);
    --   Compute the length of a given statement

    procedure RECOGNIZE_STATEMENT
    (P: in out STATEPOINT; L: in LINEPOINT);
    --   Recognize the statement from
    --   the internal line structure

    procedure GET_CHAR
    (P: in STATEPOINT; N: in NUM: out CHAR);
    --   Get a character from the given statement
    --   and column

    procedure PUT_CHAR
    (P: in STATEPOINT; N: in NUM: in CHAR);
    --   Put a character into the given statement
    --   and column

    procedure FRONT_INSERT
    (P: in out STATEPOINT; L: in STATETYPE);
    --   Insert the given statement into
    --   front of the given statement ID

    procedure REAR_INSERT
    (P: in out STATEPOINT; L: in STATETYPE);
    --   Insert the given statement into
    --   rear of the given statement ID

    underflow : EXCEPTION;

    overflow : EXCEPTION;

end STATEMENT;
```

(4). Indent Module. This module will indent each line using the line module, statement module and blank module. And the indentation policy can be decided here e.g. the size of each level, the treatment of blanks, and so on.

```
with BLANK;
with STATEMENT;
with LINE;
generic type POLICYTYPE is private;
     type CONSTRUCTTYPE is private;
package   INDENT    is
     type INDENTPOINT is access INDENTNODE;
     type INDENTNODE     is record
          level      : NUM;
          constrcut  : CONSTRUCTTYPE;
     end record;
     procedure INDENT
     (P: in STATEPOINT; L: out LINETYPE);
     --   Indent a line i.e. insert or delete blanks
     --   and make line break according to the source
     --   program syntex using the information about
     --   level and construct type and so on

     procedure GET_POLICY
     (P: in POLICYTYPE);
     --   Get the indentation and objective policies
     --   for example, each level has 3 blanks
     --   and with indentation error messages.

     procedure PUT_POLICY
     (P: out POLICYTYPE);
     --   Put the indentation and objective policies

     procedure GET_INFORMATION
     (P: in STATEPOINT; L: out STATETYPE);
     --   Get the information for indentation
     --   and level documentation

     procedure PUT_INFORMATION
     (P: in STATEPOINT; L: in STATETYPE);
     --   Put the information for indentation
     --   and level documentation
end INDENT;
```

(5). Program Module. This module will hide the program characteristics. It should be highly dependent on each programming language. It have two procedures - scanner and parser.

```
with LINE;
with STATEMENT;
with BLANK;
with SYMECLTABLE;
with LEVEL;
with STACK;
with QUEUE;
package   PROGRAMPART   is

     procedure SCANNER
     (P: in out STATEPOINT; L: out ITEMTYPE);
     --   Scan the scurce program and recognize
     --   each statement type for parser

     procedure PARSER;
     --   Recognize the construct of the source
     --   program

end PROGRAMPART;
```

(6). Master Module. This module will control all abcve mcdules.

```
with PROGRAMPART;
with INDENT;
with INPUT;
with CUTPUT;

     procedure MASTER;
     --  Control all the module for reformatting
     --  and level structure documentation
```

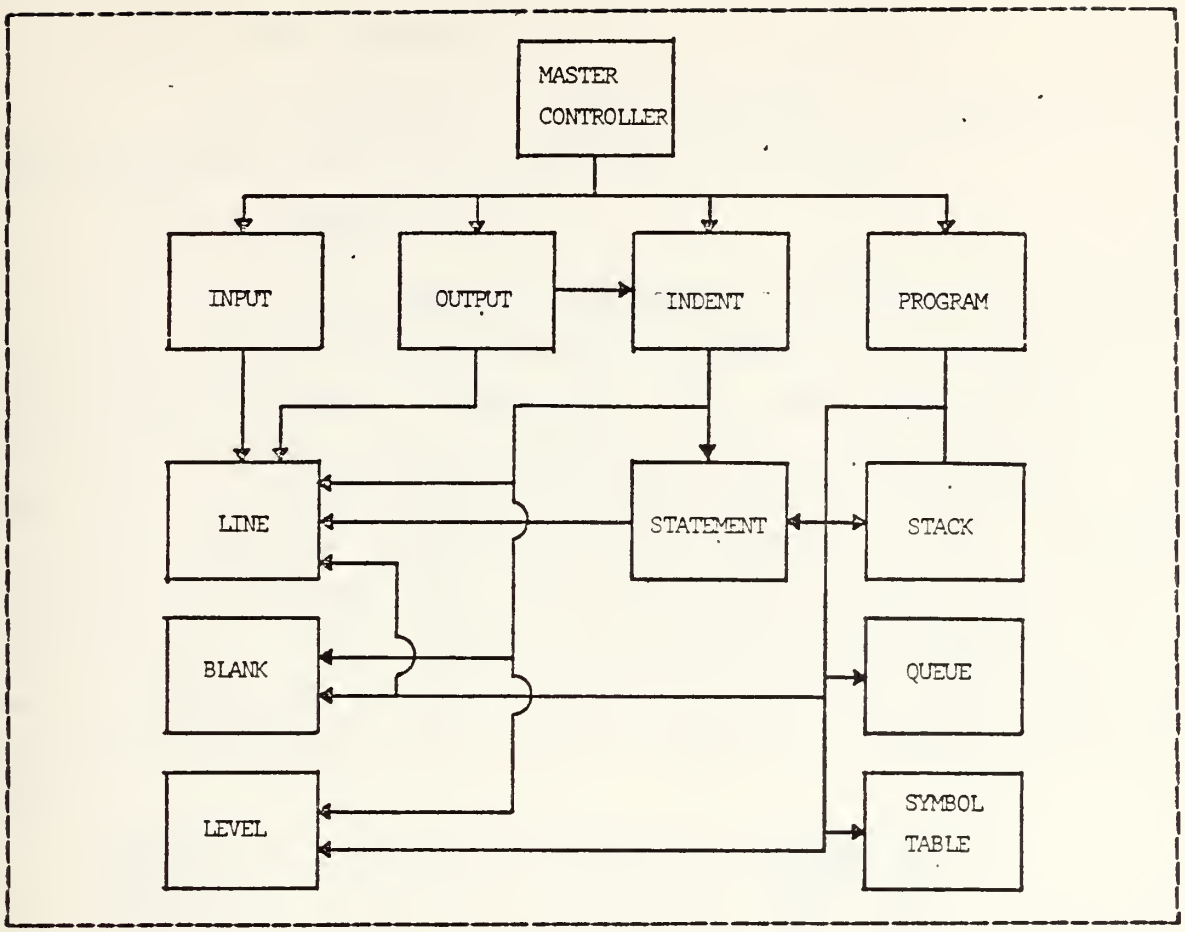

Figure 5.1 Module Interface.

The above figure explain the interfaces of each module. The arrow direction indicates using module.

D. EXAMPLE (FORTRAN)

1. Standard Form

There have been many attempts to standardize the FORTRAN programming language. Here, the standard form will follow the concept of COMPATIBLE FORTRAN [Ref. 1]. The following represent the rough standard form.

a. Basic Components

It consists of four elements - character set, symbolic names, constants and array elements.

b. Statements

(1). Statement Components. Statements are made up of such components as labels, keywords, symbolic names, constants and special characters. For Compatible FORTRAN, a stricter rule should be observed: (1). Statement labels, keywords, symbolic names, integer constants should not have embedded blanks, except for key words GO TO, DOUBLE PRECISION and BLOCK DATA, which may have blanks in the positions shown. (2). Where two alphabetic or numeric statement components come together with no other special characters between them, a blank should be inserted. Example are:

```
DO15I=1,10                                  DO 15 I=1,15
REWINDJ        should be written            REWIND J
REALAAA                                     REAL AAA
```

(3) Keywords, labels, symbolic names or constants should not be split between two lines.

(2). END Line. END is not considered a statement but is a type of line. It may not be labelled, executed or continued. Note especially that END is not an executable statement with the same effect as RETURN in a subrprogram or STOP in the main program.

(3). Format of Statements. The Standard limits each statement to one initial line and not more than 8 continuation lines.

(4). Order of Statements. The following table show the order of statements. By 'header statement' is meant a SUBROUTINE, FUNCTION or BLOCK DATA statememt. Horizontal lines within the table indicate that entities above the line must precede entities below the line (if present).

TABLE III

Table of Statement Order

Header Statement		
Type Statements		
Comment Lines	DIMENSION Statements COMMON Statements	EXTERNAL Statements
	EQUIVALENCE Statements	
	DATA Statements	
	Statement Functions	
	Executable Statements	
	STOP line	
	FORMAT Statements	
END line		

Vertical lines indicate that the entities on either side of the line may be intermingled [Ref. 1].

c. Specification Statements

Specification statements are non-executable statements which give information to the compiler. It consists of TYPE (DOUBLE PRECISION, INTEGER, REAL, LOGICAL, and COMPLEX), DIMENSION, COMMON, DATA and EQUIVALENCE.

d. Transfer of Control

This consists of the GO TO statement, Computed GO TO statement, RETURN and STOP statements, Arithemetic IF statement, Logical IF statement, DO statement, and CONTINUE statement.

e. Input/Output

This consists of the WRITE statement, READ statement, ENDFILE statement, REWIND statement, BACKSPACE statement and FORMAT statement.

f. Expression and Assignment

This consists of the Arithemetic Expression, Logical expression, and Assignment statement.

g. Program Units

This consists of the Main program, Function subprograms, Block Data, and Subroutine subprograms.

2. Structured Form

The algorithm language [Ref. 18] is convenient for representing the generalized construct structure. So, to represent the structured FORTRAN form, it will be compared with the algorithm language. Detail structured forms are as follows:

ALGORITHM LANGUAGE	FORTRAN IV
1. ALGORITHM	
ALGORITHM algorithm_name statements END algorithm_name	same_with 'C' in column 1
2. IF_THEN_single statement	

IF condition THEN
statement
END IF
	IF (condition) statement

3. IF_THEN_multiple statements	

IF condition THEN
statements
END IF
	IF (.NOT. condition) GO TO 10
	statements
	10 CONTINUE

4. IF_THEN_ELSE construct	
IF condition THEN	IF (.NOT. condition) GO TO 5

```
     statements_1                       statements_1
ELSE                                  GO TO 6
     statements_2                   5 CONTINUE
END IF                                    statement_2
                                    6 CONTINUE
```

5. Multiway selection : ELSE IF

```
IF continue_1 THEN                    IF (.NOT. condition_1) GO TO 10
     statements_1                          statements_1
ELSE IF condition_2 THEN              GO TO 20
     statements_2                   10 IF (.NOT. condition_2) GO TO 11
ELSE IF condition_3 THEN                   statements_2
     statements_3                     GO TO 20
ELSE                                11 IF (.NOT. condition_3) GO TO 12
     statements_4                          statements_3
END IF                                GO TO 20
                                    12 CONTINUE
(ELSE is optional)                         statements_4
                                    20 CONTINUE
```

6. WHILE repetion

```
WHILE condition DO                     5 IF (.NOT. condition) GO TO 6
     statements                              statements
END WHILE                                 GO TO 5
                                        6 CONTINUE
```

7. REPEAT repetition

```
REPEAT                                 5 CONTINUE
     statements                              statements
UNTIL condition                           IF (.NOT. condition) GO TO 5
```

8. DO FOR repetition

```
FOR I <- L TO M BY N DO                   DO 10 I = L,M,N
     statements                                statements
END FOR                               10 CONTINUE
(BY N can be omitted, in                  (,N can be omitted in which case
which case BY 1 is assumed)                ,1 is assumed)
```

9. Multiway selection _ CASE

```
CASE variable OF                          IF (variable.LT.1) GO TO 20
1:                                        IF (variable.GT.3) GO TO 20
     statements_1                         GO TO (11,12,13), variable
2:                                    11 CONTINUE
     statements_2                              statements_1
3:                                        GO TO 30
     statements_3                     12 CONTINUE
ELSE                                           statements_2
     statements_4                         GO TO 30
ELSE CASE                             13 CONTINUE
                                           statements_3
(ELSE is optional)                    20 CONTINUE
                                           statements_4
                                      30 CONTINUE
```

10. FUNCTION

```
FUNCTION function_name(parm_1,..., parm_n)
     statements
```

```
     function_name_expression
END function_name
```

```
          data type FUNCTION function_name(parm_1,...,parm_n)
               statements
               function_name = expression
               RETURN
          END
```

11. PROCEDURE (SUBROUTINE)

```
PROCEDURE procedure_name(parm_1,..., parm_n)
     statements
END procedure_name
```

```
          SUBROUTINE subroutine_name(parm_1,..., parm_n)
               statements
               RETURN
          END
```

3. Format Grammar

This grammar represents the construct format of structured FORTRAN. It is a subset of the generalized format structure. The control structure is limited to 5 structures - if, case, while, until, and do. In the declaration part, the declarations will be statements. For more detail, the grammer figures (Appendix B) can be referenced.

4. Implementation

a. Limitations

An ADA compiler was not available for this work. So, the PASCAL programming language was used to implement the system. This implementation is a little different from the design of the previous section because PASCAL does not support all the ADA programming features. In order to simply the implementation, just a subset of the system was implemented, i.e. the UNTIL construct is omitted.

Also the implemented system does not cover all standard FORTRAN - it does not include some keywords like PAUSE, REWIND and so on. The other limitations of this are

the fcllcwing: 1. All input programs should be syntatically correct tc get proper indentation and the level documentation. 2. All input FORTRAN programs should be conform tc the standard structured form mentioned in previous sections. 3. The input lines should be short enough to indent without being extended onto the next line. That is the implemented system does not have the line break function.

b. Internal Data Structure

(1). Line Data Structure. The input line and output line are represented as an array of characters. Normally, programming langugages use 80 column per line. In actual programs, most lines do not use all of the columns; the mean of programming line size is 34 [Ref. 2]. If the maximum array is assigned for one line, space is wasted. So to save memory and make the line flexible, a double linked data structure was used for the internal line structure. Also, a sentinel node will be used. It allows an easy check of an empty input file.

(2). Statement Data Structure. As shown above, the relationship of line and statement is one to one or many to one. Clearly, the statement can be represented by the line data structure. So, a line record will have information about statements. Comment statements will be ignored for statement representation.

(3). Construct Data Structure. The construct will have some relationship with the statements e.g. one to one for simple statements, one to many for others. The statements can have the information of the construct, since every construct can be seperated into statements. For example, the DO construct consist of DO_COND statement, compound statement and END_DO statement. But here, the line also will have the construct information. It is possible since the relationship of line and statement also one to one and many to one.

c. The Program and Example Input/Output

Anyone interested in obtaining a copy of the program should contact the author directly or the Computer Science Department at the Naval Postgraduate School, Monterey, California. The example input output can be referenced in Appendix C. The example program does not have any meaning. It is written just to show the constructs of programs and the results of program execution.

VI. CONCLUSION

One of today's software problems is the very high cost of developing and maintaining software. Much research has been devoted to solving this problem. One way to solve today's software crisis is to study software tools that can help people who serve in the software area.

This thesis designed and partially implemented a program family of extended pretty printers that can help to solve software problems by improving readability and understandability of programs.

The system will work for almost any structured programming language and for various secondary functions with only small changes in some modules. The design presented here is for a program family of pretty printers. The program implemented here is one member in this family. Other members of the program family remain to be implemented.

APPENDIX A
GENERALIZED CONSTRUCT FLOW CHART

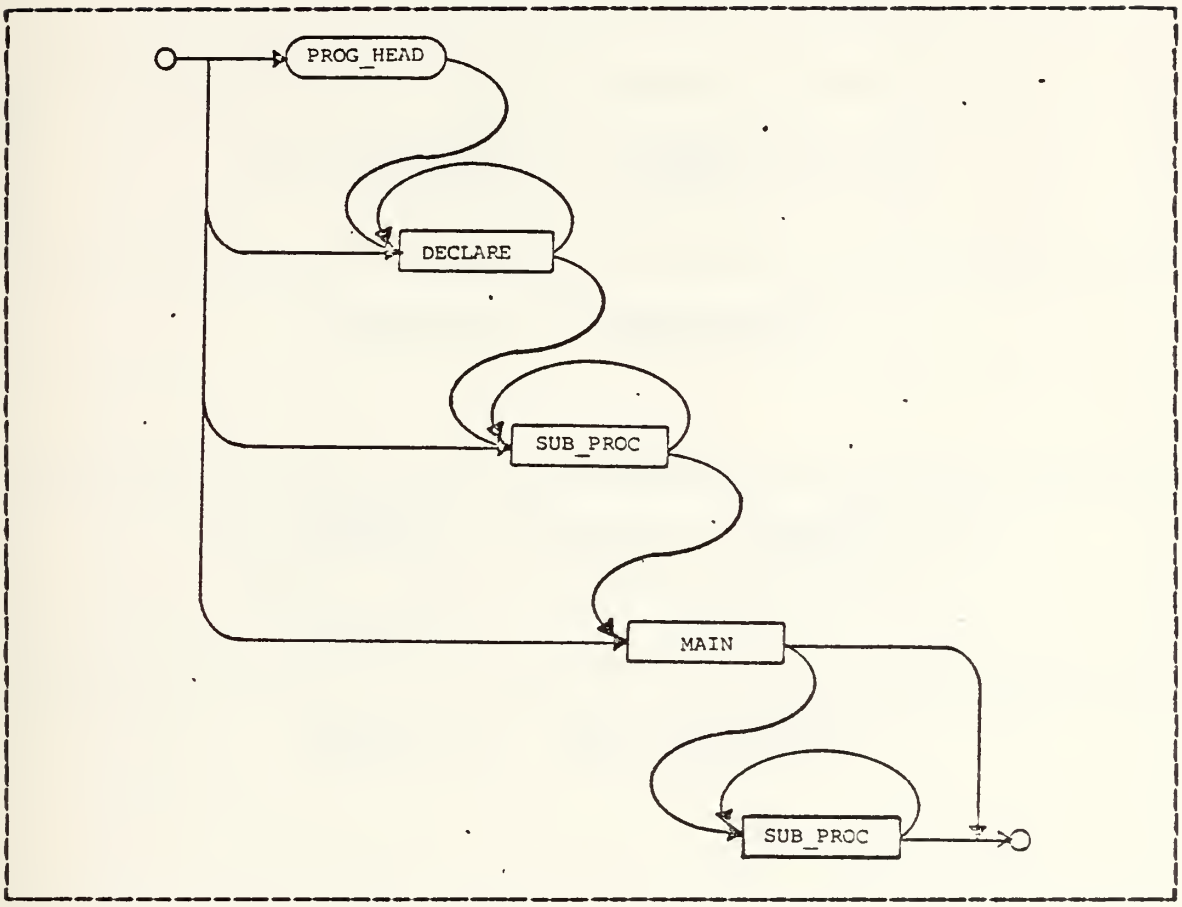

Figure A.1 Program Structure.

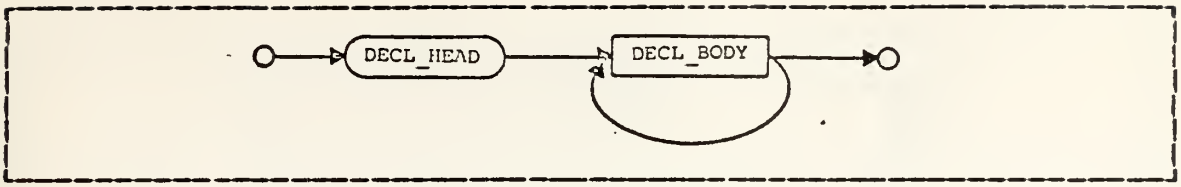

Figure A.2 Declaration.

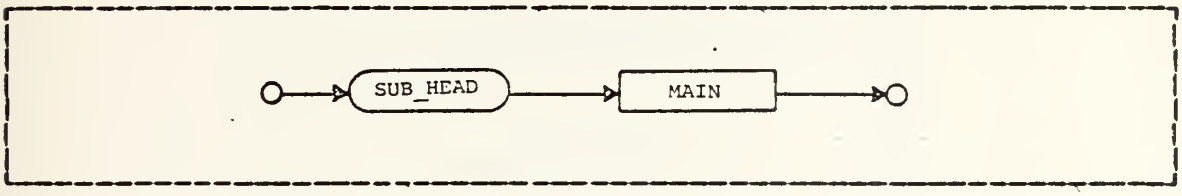

Figure A.3 Subprocedure.

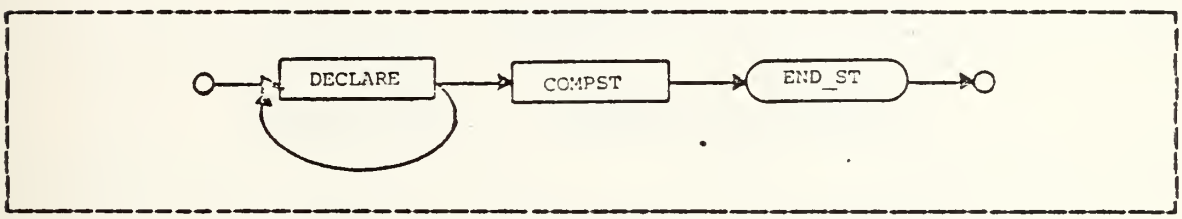

Figure A.4 Main Procedure.

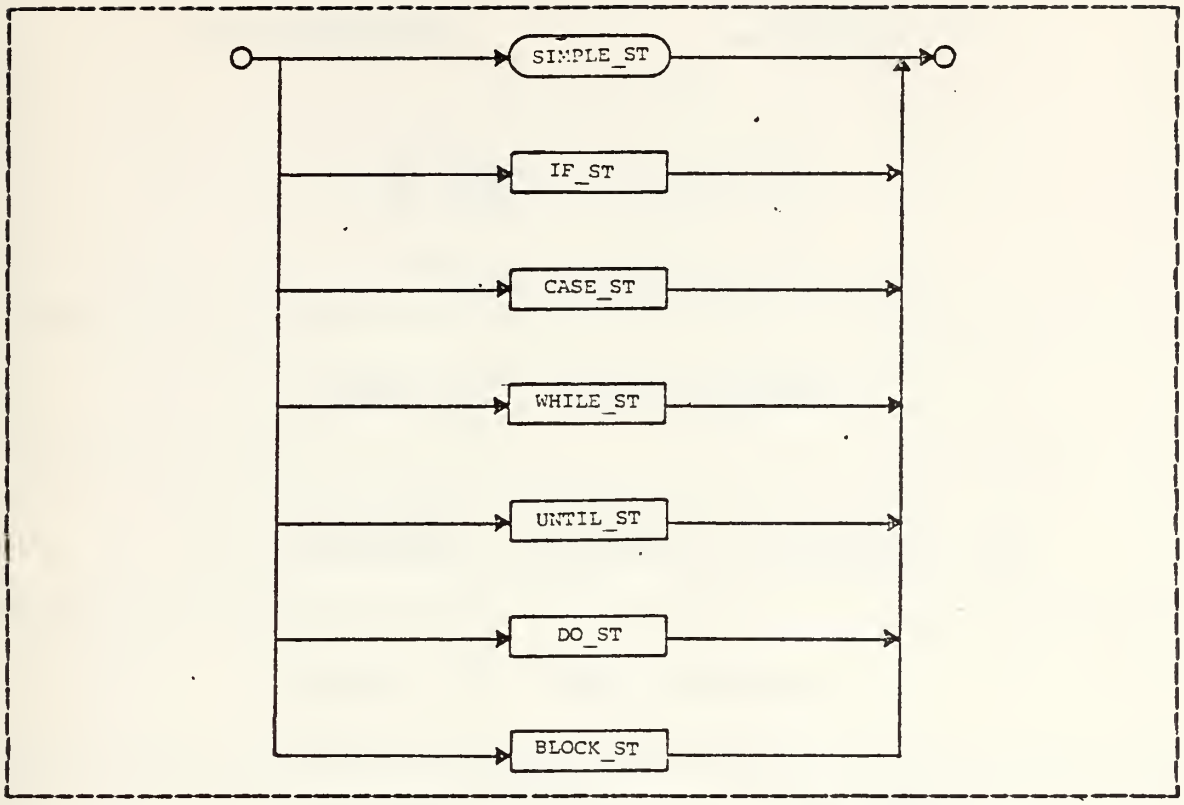

Figure A.5 Compound Statement.

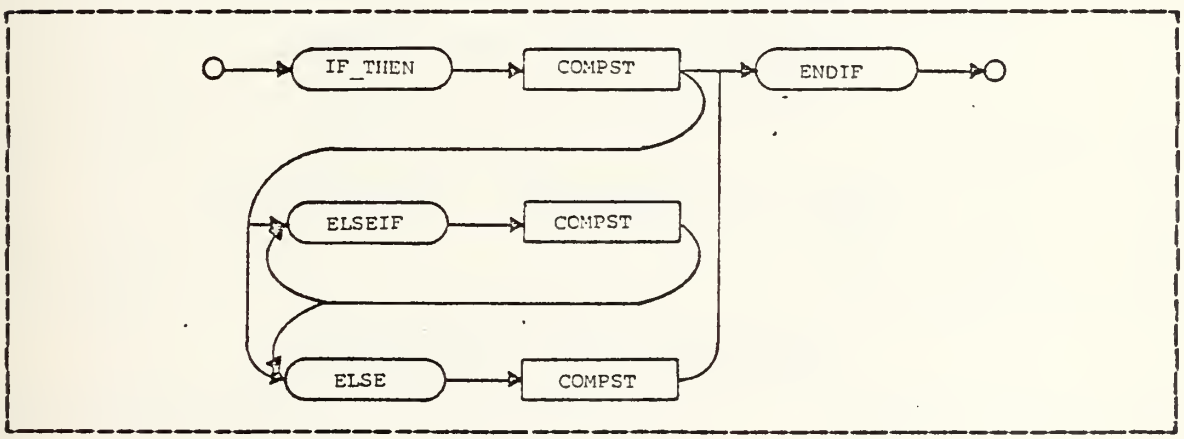

Figure A.6 If Statement.

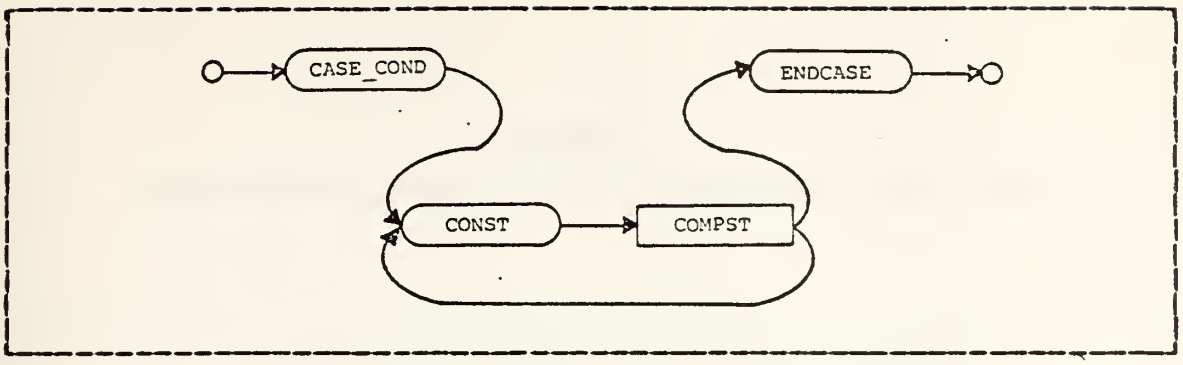

Figure A.7 Case Statement.

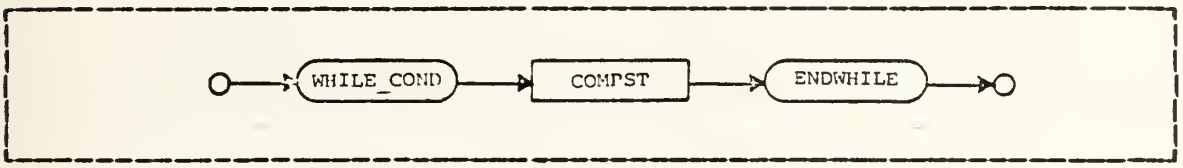

Figure A.8 While Statement.

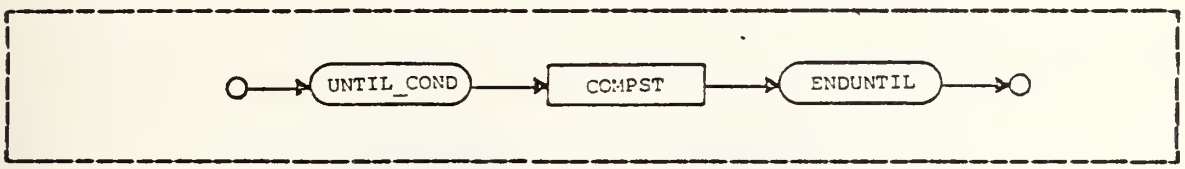

Figure A.9 Until Statement.

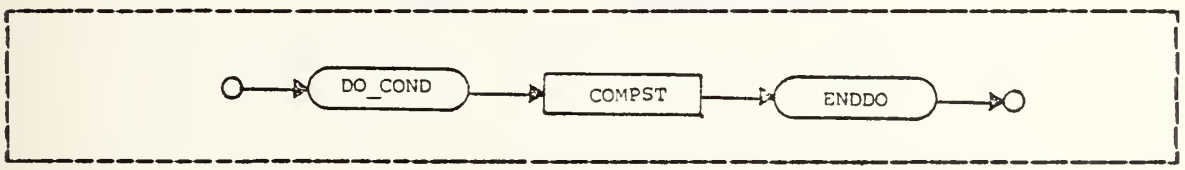

Figure A.10 Do Statement.

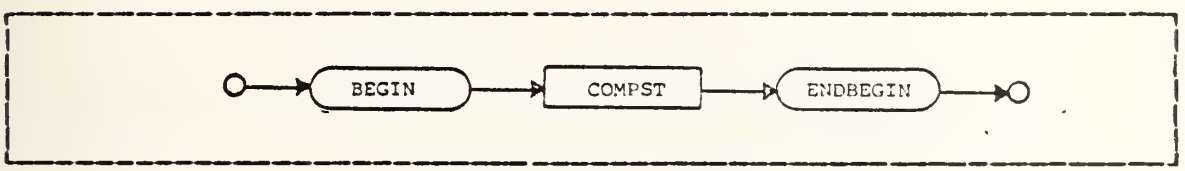

Figure A.11 Block Statement.

APPENDIX B

STRUCTURED FORTRAN FORMAT CONSTRUCT FLOW CHART

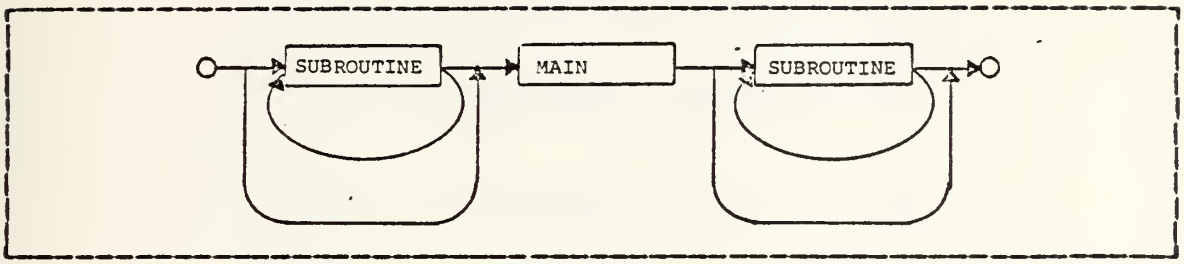

Figure B.1 Program Structure.

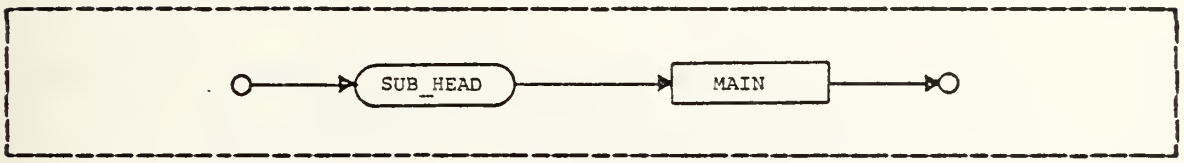

Figure B.2 Subroutine.

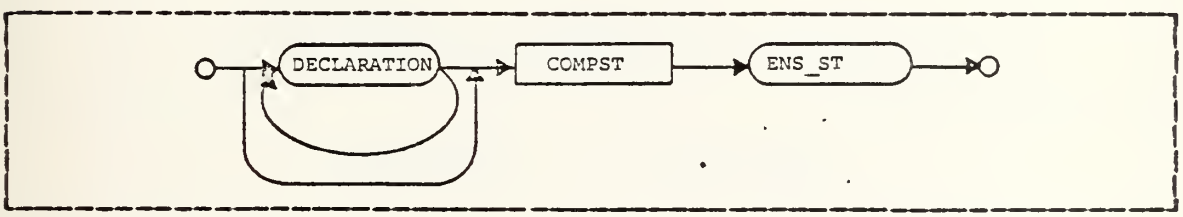

Figure B.3 Main.

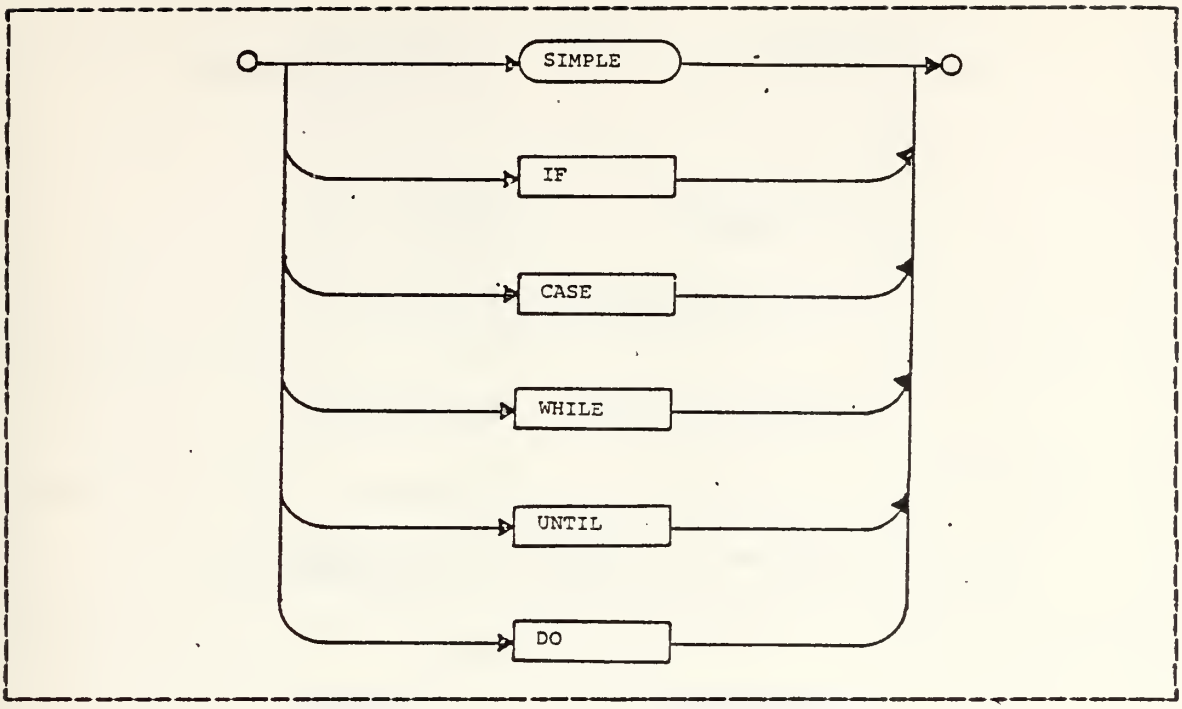

Figure B.4 Compound Statement.

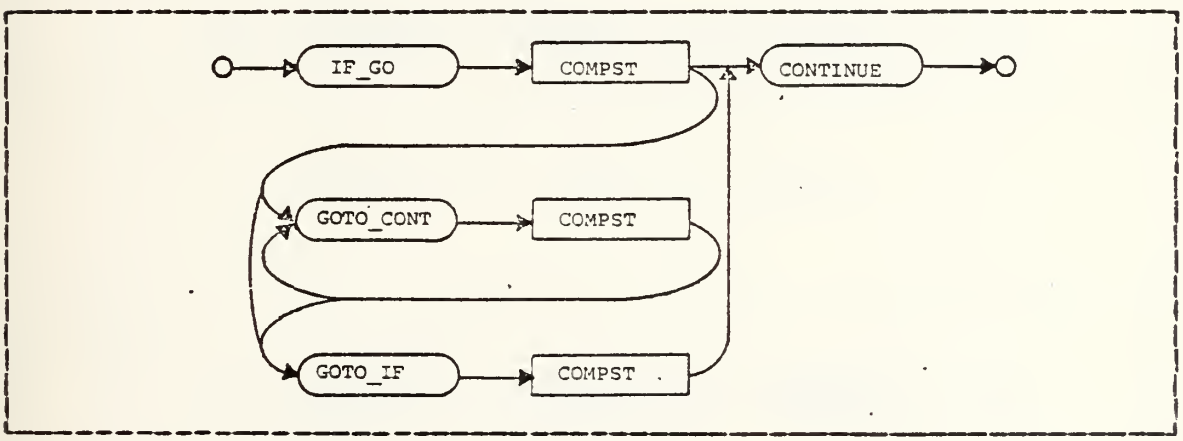

Figure B.5 If Statement.

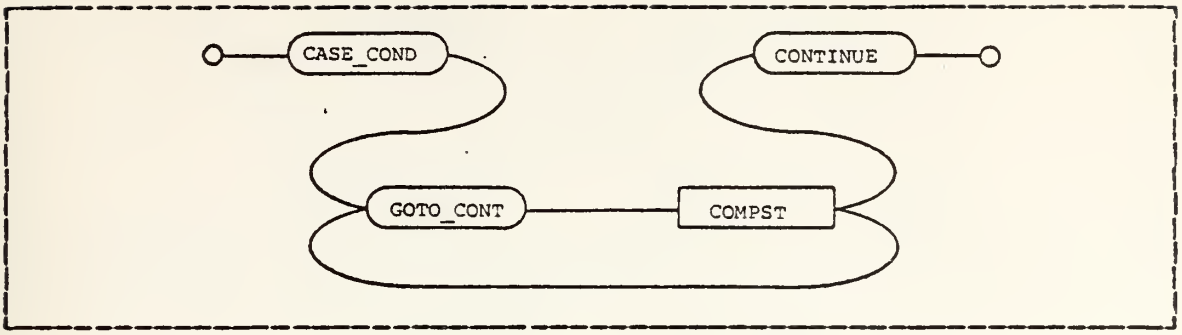

Figure B.6 Case Statement.

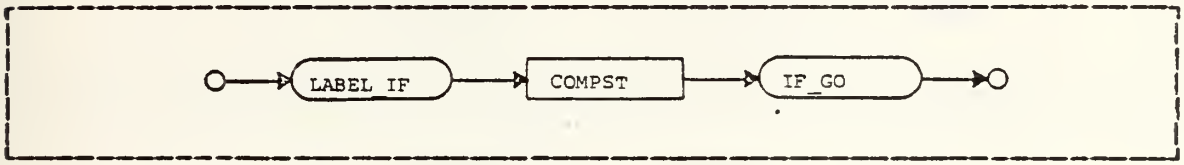

Figure B.7 While Statement.

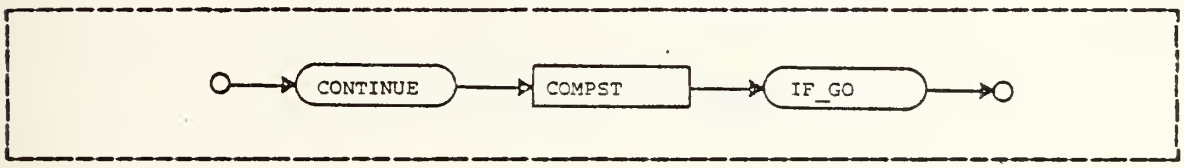

Figure B.8 Until Statement.

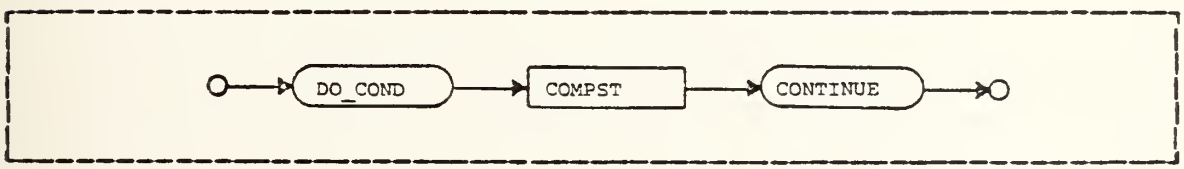

Figure B.9 Do Statement.

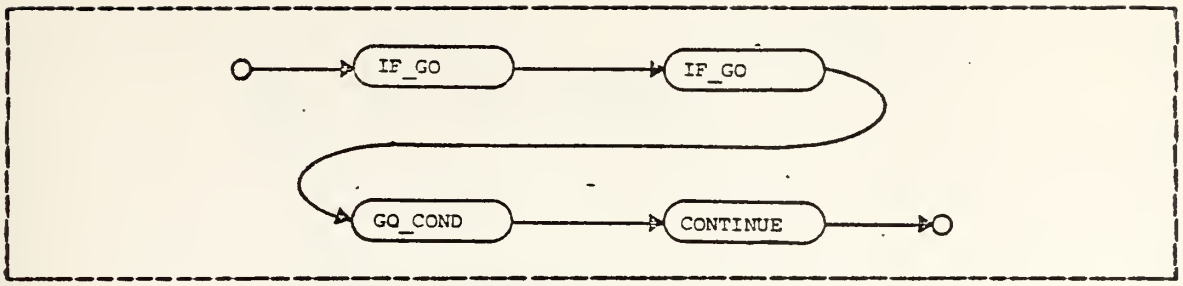

Figure B.10 Case_Cond.

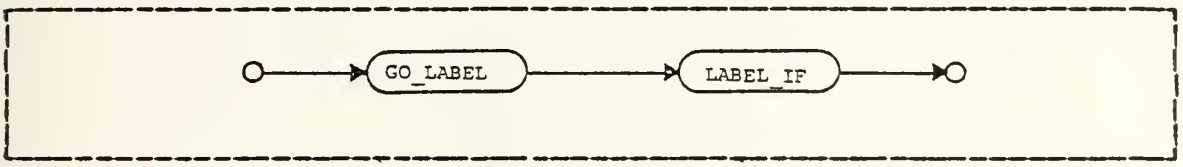

Figure B.11 Go_If.

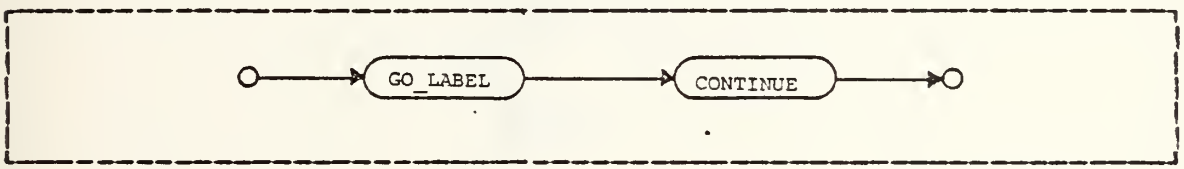

Figure B.12 Go_Cont.

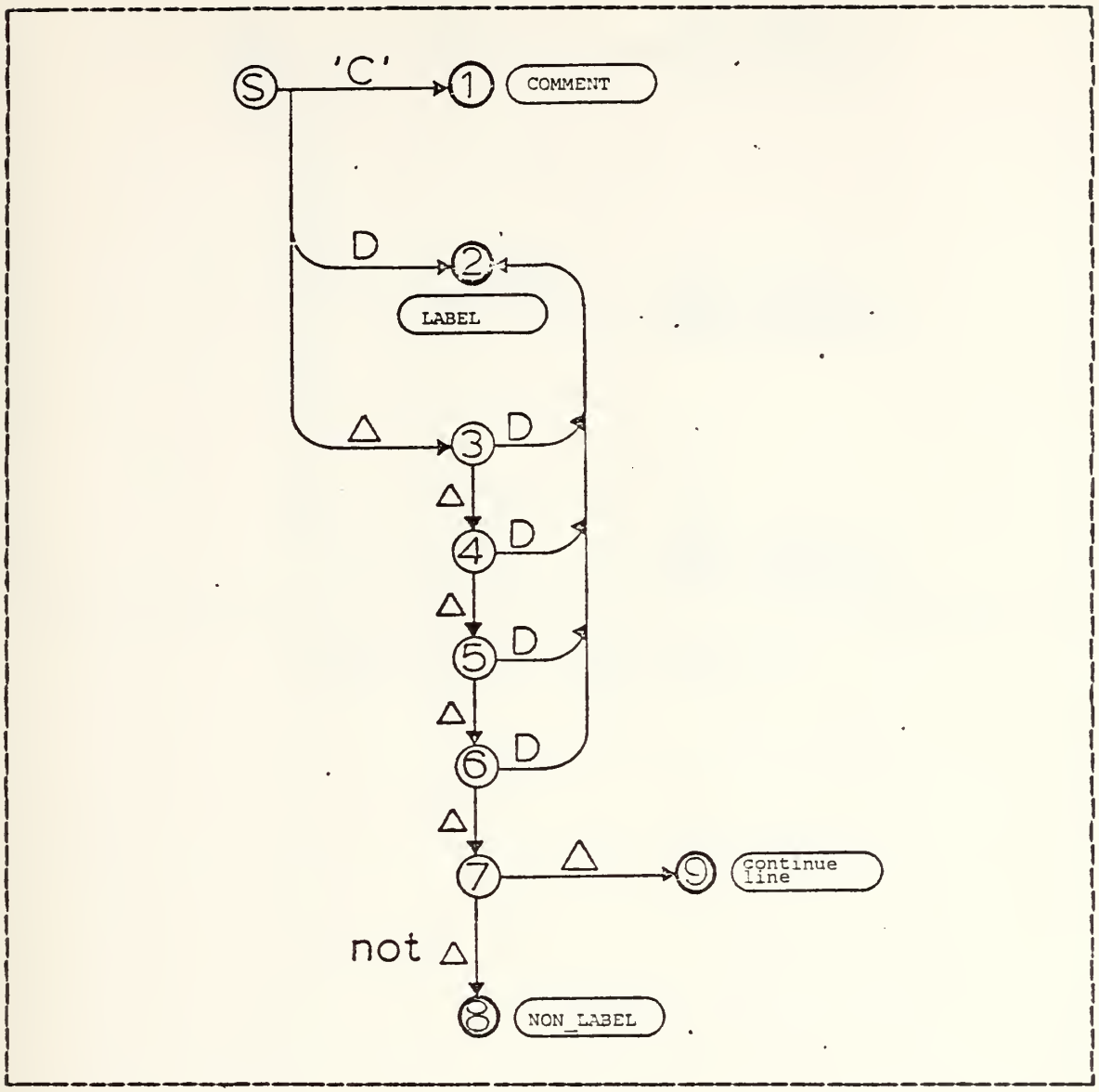

Figure B.13 State Chart 1.

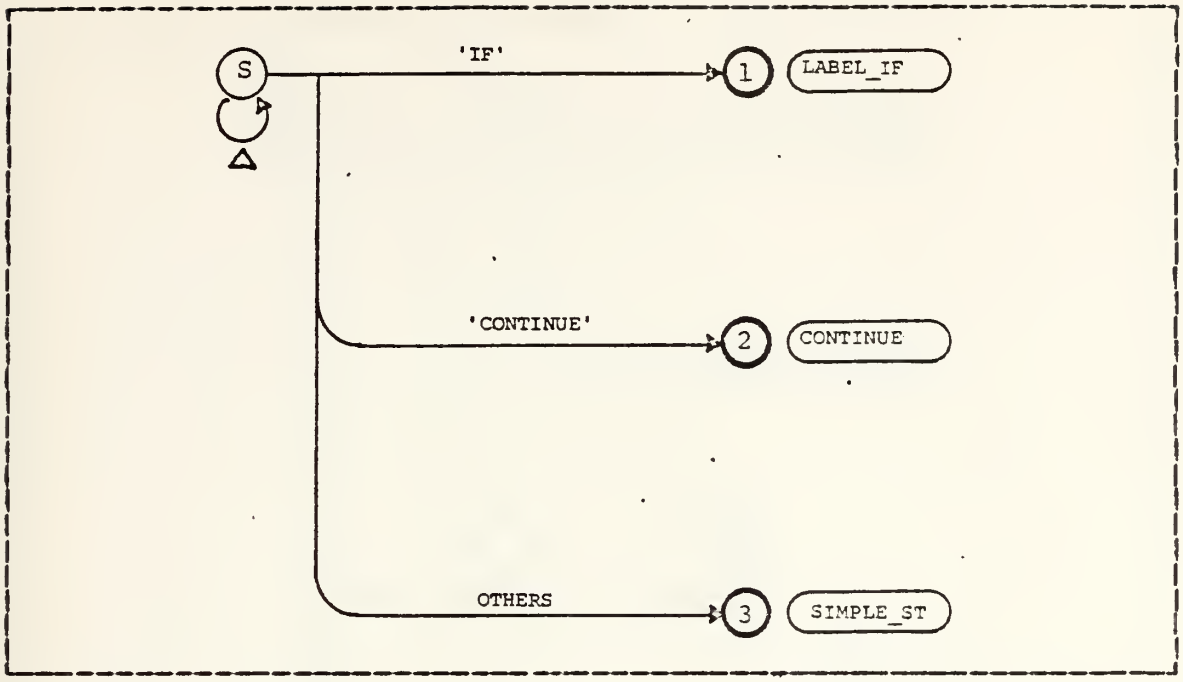

Figure B.14 State Chart 2.

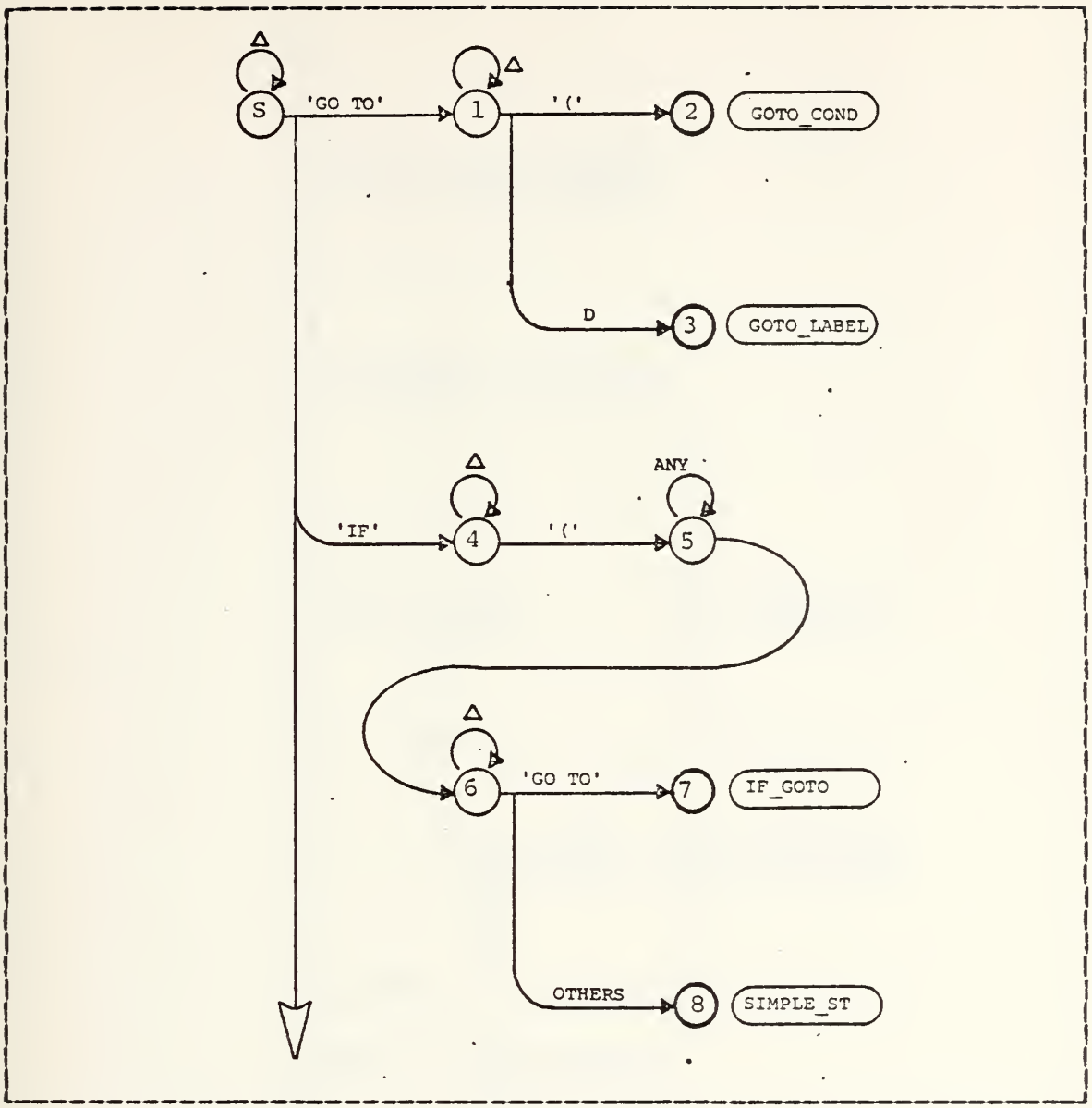

Figure B.15 State Chart 3.

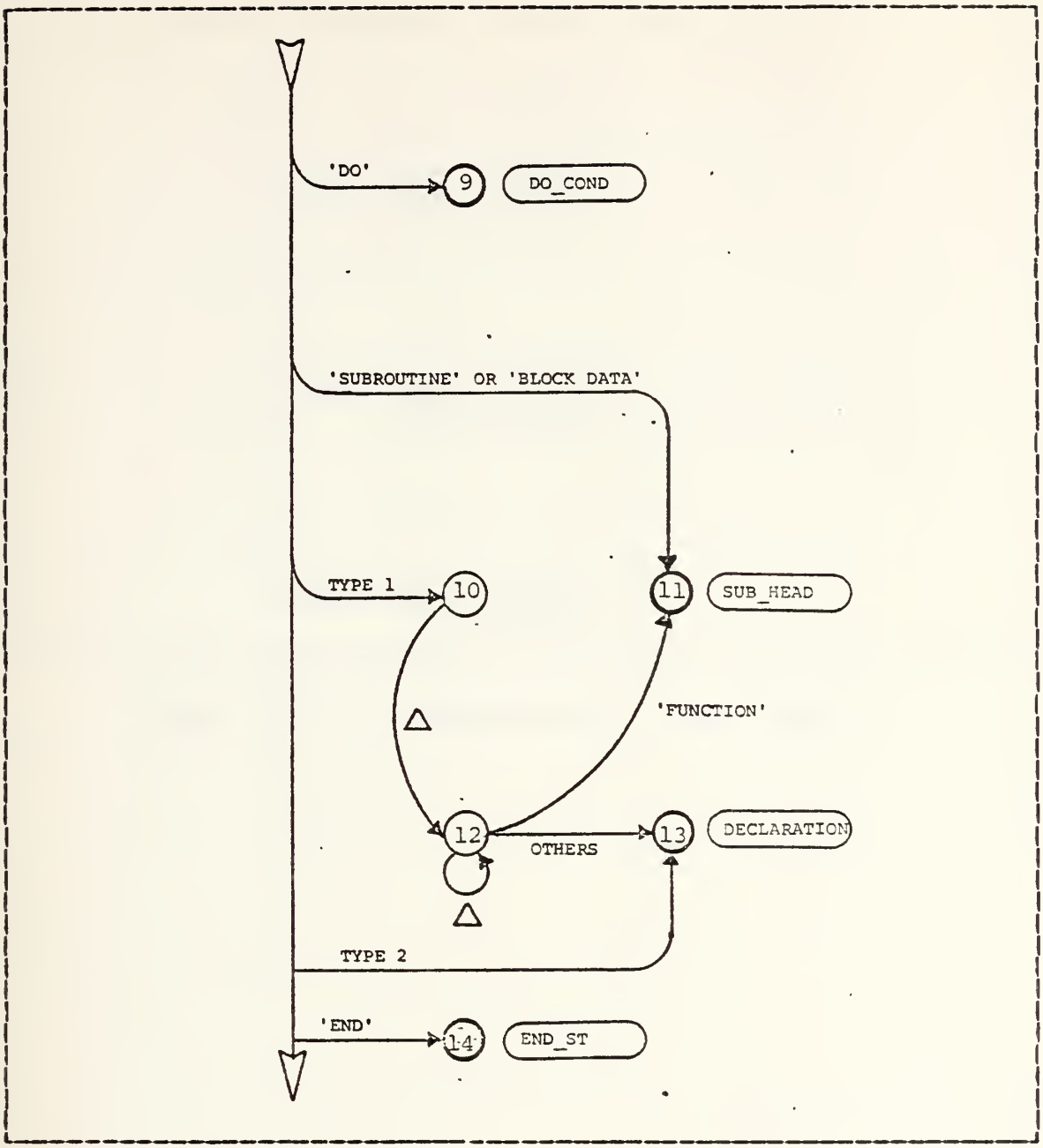

Figure B.16 Continuation of State Chart 3.

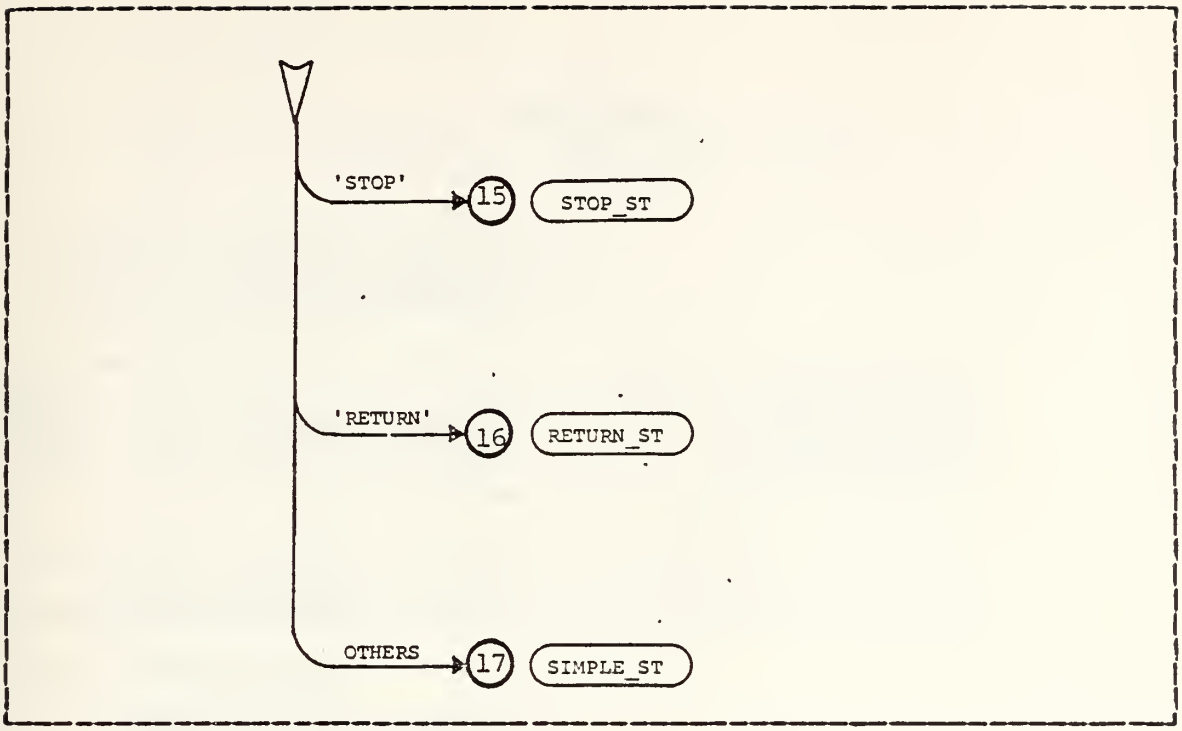

Figure B.17 Continuation of State Chart 3.

APPENDIX C

EXAMPLE OF INPUT AND OUTPUT

```
   ============================
    *** INPUT NOT INDENTED ***
   ============================


C*************************************************
C***                                           ***********
C***   TEST PROGRAM FOR THE AUTOMATIC          ***********
C***   INDENTATION PRCGRAM                     ***********
C***                                           ***********
C*************************************************
C
C
C*************************************************
C      MAIN PROGRAM
C*************************************************
C
C***   DECLARATION
C
       REAL R1,R2,R3,RD(20)
       INTEGER I1,I2,I3,ID(20)
       LOGICAL L1,L2,L3
C
C***   CCMPCUND STATEMENT
C
C          SIMPLE STATEMENT
C          IF STATEMENT
C          CASE STATEMENT
C          WHILE STATEMNET
C          DO STATEMENT
C
C
       READ(5,100) I1,I2,I3
   100 FORMAT(3I5)
       L1 = .TRUE.
       R1 = 1.5
      1+5.0-6.7
      2+4.8
       CALL SUB(RD,ID)
       IF(.NOT.(I1.NE.1)) GO TO 1
       DO 500 I = 1,20
       R(I)=0.0
   500 CONTINUE
       GO TO 444
     1 IF(.NOT.(I2.NE.2)) GO TO 2
   111 IF(.NOT. L1) GO TO 11
       I1=I1+1
       GO TO 111
    11 CONTINUE
       GO TO 444
     2 IF(I3.NE.3) GO TO 3
       I2=I2+1
       GO TO 444
     3 CONTINUE
       I3=I3+1
   444 CONTINUE
       R(I1)=5.5
       IF(I1.LT.5) GO TO 555
       IF(I1.GT.3) GO TO 555
```

```
      GO TO (5,6,7)
    5 CONTINUE
      RD(1)=5.0
      RD(2)=5.0
      GO TO 666
    6 CONTINUE
      RD(1)=6.0
      RD(2)=6.0
      GO TO 666
    7 CONTINUE
      DO 567 I=1,19
      RD(I)=FLOAT(I)
  567 CONTINUE
      RD(20)=40.0
  555 CONTINUE
      R1=4.9
      I1=4*I2
      STOP
      END
C
C
C
C
C*************************************************
C     SUBROUTINE PROGRAM
C*************************************************
      SUBROUTINE SUB(RD,ID)
C
C***  DECLARATION
C
      REAL R1,R2,R3,RD(20)
      INTEGER I1,I2,I3,ID(20)
      LOGICAL L1,L2,L3
C
C***  SIMPLE STATEMENT
C
      READ(5,100) I1,I2,I3
  100 FORMAT(3I5)
      L1 = .TRUE.
      R1 = 1.5
C
C*** IF STAIEMENT
C*** DO STATEMENT
C
      IF(.NOT.(I1.NE.1)) GO TO 1
      DO 500 I = 1,20
      R(I)=0.0
    1 CONTINUE
      RETURN
      END
```

```
======================
 *** END OF INPUT ***
======================
```

```
==========================
 *** PROGRAM CONSTRUCT ***
==========================

 DECLARATICN
 DECIARATICN
 DECLARATION
      SIMPLE
      SIMPLE
      SIMPLE
      SIMPLE
      SIMPLE
      IF (COND) THEN
           DO (COND)
                SIMPLE
           END DO
      ELSE IF (COND)
           WHILE (CCND) DO
                SIMPLE
           END WHILE
      ELSE IF (COND)
           SIMPLE
      ELSE
           SIMPLE
      END IF
      SIMPLE
      CASE  VAR
      CCNST :
           SIMPLE
           SIMPLE
      CCNST :
           SIMPLE
           SIMPLE
      CCNST :
           DO (COND)
                SIMPLE
           END DO
           SIMPLE
      END CASE
      SIMPLE
      SIMPLE
 STOP
 END OF PROGRAM
 SUBROUTINE
 DECLARATICN
 DECLARATICN
 DECLARATION
      SIMPLE
      SIMPLE
      SIMPLE
      SIMPLE
      IF (COND) THEN
           DO (COND)
                SIMPLE
           END DO
 RETURN
 END OF PROGRAM

==========================
 *** END OF CONSTRUCT ***
==========================
```

```
   ==========================
    *** OUTPUT INDENTED ***
   ==========================


C*************************************************
C***                                            ***********
C***   TEST PROGRAM FOR THE AUTOMATIC           ***********
C***   INDENTATION PROGRAM                      ***********
C***                                            ***********
C*************************************************
C
C
C*************************************************
C     MAIN PROGRAM
C*************************************************
C
C***  DECLARATION
C
      REAL R1,R2,R3,RD(20)
      INTEGER I1,I2,I3,ID(20)
      LOGICAL L1,L2,I3
C
C***  COMPOUND STATEMENT
C
C         SIMPLE STATEMENT
C         IF STATEMENT
C         CASE STATEMENT
C         WHILE STATEMNET
C         DO STATEMENT
C
C
            READ(5,100) I1,I2,I3
  100       FORMAT(3I5)
            L1 = .TRUE.
            R1 = 1.5
        1   +5.0-6.7
        2   +4.8
            CALL SUB(RD,ID)
            IF(.NOT.(I1.NE.1)) GO TO 1
                 DO 500 I = 1,20
                      R(I)=0.0
  500            CONTINUE
            GO TO 444
    1       IF(.NOT.(I2.NE.2)) GO TO 2
  111            IF(.NOT. L1) GO TO 11
                      I1=I1+1
                 GO TO 111
   11            CONTINUE
            GO TO 444
     2      IF(I3.NE.3) GO TO 3
                 I2=I2+1
            GO TO 444
     3      CONTINUE
                 I3=I3+1
  444       CONTINUE
            R(I1)=5.5
            IF(I1.LT.5) GO TO 555
            IF(I1.GT.3) GO TO 555
            GO TO (5,6,7)
     5      CONTINUE
                 RD(1)=5.0
                 RD(2)=5.0
            GO TO 666
     6      CONTINUE
                 RD(1)=6.0
                 RD(2)=6.0
            GO TO 666
     7      CONTINUE
```

```
                DO 567 I=1,19
                     RD(I)=FLOAT(I)
  567           CONTINUE
                RD(20)=40.0
  555       CONTINUE
            R1=4.9
            I1=4*I2
        STOP
        END
C
C
C
C
C*************************************************
C       SUBROUTINE PROGRAM
C*************************************************
        SUBROUTINE SUB(RD,ID)
C
C***    DECLARATION
C
        REAL R1,R2,R3,RD(20)
        INTEGER I1,I2,I3,ID(20)
        LOGICAL L1,L2,I3
C
C***    SIMPLE STATEMENT
C
            READ(5,100) I1,I2,I3
  100       FORMAT(3I5)
            L1 = .TRUE.
            R1 = 1.5
C
C*** IF STATEMENT
C*** DO STATEMENT
C
            IF(.NOT.(I1.NE.1)) GO TO 1
                DO 500 I = 1,20
                     R(I)=0.0
    1           CONTINUE
        RETURN
        END

  ========================
   *** END OF OUTPUT ***
  ========================
```

LIST OF REFERENCES

1. Day, Colin A., Compatible Fortran, Cambridge University Press, 1978

2. Bruell, S.C. and Schneider, G.M., Advanced Programming and Problem Solving with PASCAL, Wiley, 1981

3. Booch, G., Software Engineering with ADA, Benjamin/Cummings, 1983

4. Elshoff, J.L. and Marcotty, M., "Improving Program Readability to Aid Modification", Communication of the ACM August 1982 Vol. 25 No.8 pp 512-521

5. Elshoff, J.L., "An Analysis of Some Commerical PL/I Programs", IEEE Trans. Software Eng. SE-2,2, pp 113-120

6. Lientz,B.P. and Swanson,E.B., Software Maintenance Management, Addison-Wesley,Reading Mass. 1980

7. Sheppard,S.B., Curtis,B., and Milliman,P., "Modern Coding Practices and Programmer Performance", IEEE Computer, Dec. 1979 pp 41 - 49

8. Kernighan,B.W., and Plauger,P.J., The Elements of Programming Style, McGraw-Hill, New York, 1974.

9. Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, Englewood-Cliffs, N.J. 1975

10. Myers,G.J., Software Reliability, John Wiley, New York, 1976

11. Shooman, Martin L., Software Engineering, McGraw-Hill, 1983

12. Lee, Godfrey and others, "FORTRAN Programming Standards", SIGPLAN Notices 15:2 (Feb. 1980) pp 52 - 63

13. Parnas, David L., "On the Design and Development of Program Families", IEEE Transactions on Software Engineering pp 1-9, 1976

14. Rubin, Lisa F., "Syntax-Directed Pretty Printing - A First Step Towards a Syntax - Directed Editor", IEEE Tran. Software Eng. Vol.SE.9 NO.2 March 1983

15. Sheil,B.A., "The Psychological Study of Programming", Computer Surveys, Vol.13,No.1, March 1981

16. Dijkstra, E.W., "Go To Statement Considered Harmful", Communication of the ACM, Vol.15 No.10(Oct. 1972) pp 859-66

17. Shneiderman,Ben, Software Psychology, Winthrop Publishers, 1980

18. Graham,N., Introduction to Computer Science, West Publishing Co. 1982

19. Conrow,K. and Smith, R.G., "Neater2: A PL/I Source Statement Refcrmater", Communication of the ACM 13,11(Nov. 1970), pp 669-675

20. Condict,M.N., Marcus,R.L. and Mickel,A., "Pascal Prcgram Formatter", Pascal News No.13 (Dec. 1978) pp 45 - 58

21. Hueras,Jon F. and Ledgard, Henry F., "Prettyprint", Pascal News No. 13 (Dec. 1978) pp 34 - 44

22. Gimpel, James F., "CONTOUR - A Method of Preparing Structured Flowcharts", SIGPLAN Notices Vol.15 No. 10 (Oct. 1980) pp 35 - 41

23. Barnes,J.G.P., Programming in ADA, Addison-Wesley Publishing Co., 1982

24. Maych,Brian, Problem Solving with ADA, John Wiley & Sons, 1982

25. Gehani, Narain, ADA an Advanced Introduction, Prentice-Hall Software Series, 1983

26. Shneiderman, B., "Exploratory Experiments in Programmer Behavior", Int. J. Comput. Inf.Sci. 5 (1976), pp 123-143

27. Chase,W.G. and Simon,H.A., "Perception in Chess", Ccgnitive Psychol. 4 (1973), pp 55-81

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Professor Gordon Hoover Bradley Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
4.	Professor Bruce James MacLennan Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Naval Postgraduate School Computer Technology Curricular Office Code 37 Monterey, California 10996	1
6.	Department Chairman Department of Computer Science Code 52 Naval Postgraduate School Monterey, California 93940	1
7.	CPT Dan P. Krebill Computer Systems Division DAAS, Thayer Hall West Point, New York 10996	1
8.	CPT Mark R. Kindl 413 E. Washington ST. Villa Park, IL 60181	1
9.	Capt T. F. Rogers Jr. BOX 327 Lumberport WV 26386	1
10.	Major Tae Nam Ahn Computer Center P.O. BOX 77 GongNeung Dong, Dobong_Gu Seoul 130-09, Korea	4
11.	Library P.O. BOX 77 GongNeung Dong, DoBong_Gu Seoul 130-09, Korea	1
12.	Lt. Col. C.I. Park SMC 2139 NPGS Monterey CA 93940	1

13. Major K.S. Bak 1
SMC 1034 NPGS
Monterey CA 93940

14. Major B.N. Ko 1
SMC 2769 NPGS
Monterey CA 93940

15. Major K.J. Nam 1
SMC 2817 NPGS
Monterey CA 93940

16. Major M.T. Seo 1
SMC 2768 NPGS
Monterey CA 93940

17. Major K.H. Chung 1
SMC 1349 NPGS
Monterey CA 93940

18. Major S.K. Kim 1
SMC 2793 NPGS
Monterey CA 93940

19. Major J.H. Kim 1
SMC 1308 NPGS
Monterey CA 93940

20. Lt. Col. S.H. Cha 1
SMC 2945 NPGS
Monterey CA 93940

21. LCDR Mark J. Geschke 1
Damage Control Assistant
USS Midway (CV 41)
FPO San Francisco, CA 96601